UiO : Department of Informatics
University of Oslo

Storage of Genomic Data using PyTables

Brynjar G. Rongved

Master's Thesis Autumn 2014

Abstract

The later years have seen an increasing demand for systems that can perform fast genome-wide analyses. An important component of such a system is the data model. The data model of the Genomic HyperBrowser analysis system has recently been extracted to a package called GTrackCore. This package is planned to be integrated in a standalone command-line based analysis toolset.

The data produced by GTrackCore is currently stored in an ad-hoc way. Due to some problems with this storage method, it would be beneficial to replace it. The proposed solution is to utilise PyTables, a package for managing hierarchical data sets, built on the HDF5 library.

This thesis presents an implementation of a PyTables-based preprocessor in the GTrackCore package. The implementation shows that PyTables can be successfully incorporated in GTrackCore without having to completely restructure the package, but that further adaptation would be beneficial.

Measurements of performance and storage efficiency show that the PyTables-based preprocessor demonstrates better or equal performance compared to the old preprocessor in most cases. Further, the PyTables-implementation solves the problems of the current ad-hoc format.

Preface

The target audience of this thesis are other master students with a background in informatics. Despite the fact that this thesis concerns contributions to a package that probably will be used in biological research in the future, any knowledge of biology is not required.

On collaboration with master student Henrik Glasø Skifjeld, and division of thesis focus

The bulk of the work that led up to this thesis is the result of a close collaboration with another master student. The work has involved a comprehensive refactoring of a software package used for managing data related to genomes. We have replaced functionality in the two main components of the package. One of these components is responsible for the creation and storage of binary data, while the other is responsible for the retrieval of the same data. Due to the fact that the majority of the development has been conducted in close collaboration, we have an almost equal level of understanding of both these two components of the package.

Because there were interesting research questions related to both components of the package, we decided to let the components themselves serve as a basis for the division of thesis focus. We divided it such that I was to be responsible for problems related to storage, whereas Henrik Glasø Skifjeld was to be responsible for problems related to retrieval.

Storage and retrieval are two sides of the same coin, and it is sometimes necessary to speak about both concepts. Hence, I will at times use examples and material that perhaps can be said to be more coherent with the thesis about retrieval. I will also refer to the thesis about retrieval when I need to.

Because our theoretical basis is the same, there should be some similarities in how we have laid out the background material. The process and practices we utilised along the way are also common, and the coverage of this in the theses will therefore be similar. Although we have exchanged thoughts and ideas along the way, both theses have been written individually in their entirety.

Acknowledgements

First and foremost, I would like to thank Henrik Glasø Skifjeld, for a close and fruitful collaboration. Special thanks to my supervisors Geir Kjetil Sandve and Sveinung Gundersen for guidance and valuable feedback, which considerably improved this work.

Fredrik Valdmanis and Torkil Vederhus deserve a big thanks for proofreading parts of my thesis and provide me with useful comments.

I would also like to thank my family and friends for moral support, and my dear Therese for her love and encouragement.

Brynjar Grønhaug Rongved
University of Oslo
July, 2014

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The cost of sequencing a human genome is as low as ever before. The demand for genome-wide analyses is consequently increasing and there is a constant need for better-performing analysis software. A very important component to how such software performs is the data model. The term data model embraces all components of a computer system that are related to how the data is organised and represented, both logically and physically.

The GTrackCore package is an extraction of the data model integrated in the statistical analysis system The Genomic HyperBrowser. It contains functionality related to both storage and retrieval of data. The motivation for the extraction was for one to easily be able to integrate the model with a standalone, command-line based, analysis toolset. A second motivational factor was to have the data model of the HyperBrowser as a loosely coupled package instead of a tightly integrated module.

GTrackCore is used to store data related to genomes, where the data consist of rows representing informational elements and columns represent various aspects of each such element. GTrackCore currently uses a binary data format where storage of such data sets, referred to as *genomic tracks*, is based on NumPy memmaps, where each column of a track is stored in a separate file. There are some problems related to this data format. Due to the likelihood of genomic tracks consisting of numerous columns, the number of files per track can be high. In fact, so high that the operating system might run out of inodes, or equivalent structures of non-UNIX file systems, if enough tracks are exported to the binary data format. The data format is also not suitable for distribution. The memmaps of a given genomic track belong together, and have to be distributed together for the track to be used without loss of data. The data format does not express these dependencies between files, so additional directory or archive structures have to be added for the format to explicitly express this. The current version of the popular analysis framework Galaxy, for example, does not support upload of multi-file data sets, hence making it difficult to develop Galaxy tools that uses binary data produced by the preprocessor of GTrackCore/HyperBrowser.

The proposed solution to these problems is the Hierarchical Data Format,

or HDF5, managed through the high-level Python package PyTables. The capability of the format to store multiple data sets in the same file, along with its reputation for being an efficient and flexible file format for managing very large data sets, makes it a possible candidate for replacing the existing ad-hoc data format.

1.2 Goals

The major goal of this thesis has been to find out whether the PyTables package is suitable for storage of genomic data in GTrackCore. The practical method that has been used to achieve this goal is an implementation where PyTables was attempted to be incorporated in GTrackCore. The development of this PyTables-based GTrackCore has involved a refactoring of the inner workings of both the storage and retrieval modules. This thesis will, however, only cover the storage module. Details about the work on the retrieval module are covered by Skifjeld [38].

The PyTables package has to fulfill certain requirements in order to be classified as suitable for this project. First, the implementation has to solve the aforementioned problems related to extensive use of files to represent a single data set, and the associated problems of distribution. Secondly, the PyTables-based implementation has to be able to compete with the old memmap-based one in terms of performance and file format efficiency.

1.3 Research questions

This thesis will try to answer the following research questions.

1. Is PyTables, and the underlying HDF5 format, suitable for storage of genomic tracks in GTrackCore, and will PyTables solve the problems related to multiple files being needed to represent each individual data set?

2. Can PyTables be used for storage of genomic data in GTrackCore as it stands, or should GTrackCore be adapted to it through a comprehensive code restructuring?

3. What are the advantages and disadvantages of using a popular package such as PyTables, which is built upon the de facto HDF5 format, for storage of genomic data, compared to using an ad-hoc format such as the custom memmap-based format created for GTrackCore?

1.4 Chapter overview

Chapter 2 presents the background material considered to be necessary for understanding the rest of the thesis.

Chapter 3 gives an overview of the practices and strategies that were utilised during the development of a PyTables-based GTrackCore.

Chapter 4 presents the implementation details about how the preprocessor of GTrackCore was adapted to use PyTables.

Chapter 5 presents the results related to run time performance and efficiecy of new HDF5 file format.

Chapter 6 discusses the results. It also discusses the method that was used to ensure that the implementation is reliable, and gives an answer to the research questions.

Chapter 7 presents some ideas for areas of future work.

Chapter 2

Background

This chapter will present the theoretical foundation that the rest of the thesis builds upon. As rudimentary knowledge of genetics probably is required in order to fully understand the motivation behind this project, we will start by giving a introduction to the biological domain, by briefly explaining how DNA works, how DNA sequences are obtained, and how the underlying informational content is annotated and stored in what are known as *genomic tracks*. Further, we describe the analysis system *the Genomic HyperBrowser*, and an *abstract methodology* that the system utilises to do analyses. We will then outline the *representational formats* that are used to store the genomic tracks. This is followed by an overview of the *technologies* that we consider to be relevant. Finally, we give an introduction to the pertinent *GTrackCore* package that the main implementation of this project revolves around.

2.1 DNA

The Genetic information of any organism is stored in deoxyribonucleic acid (DNA), organised in structures called chromosomes. The DNA in all of the chromosomes combined together is the entirety of an organism's hereditary information, and is called the genome. DNA is a large molecule, a polymer, composed of four different nucleic acid units named nucleotides. Each nucleotide consists of three different parts: a base molecule, a pentose, and one or more phosphate groups. The nucleotides are often simply referred to as bases, since this is the only constituent that separates the four. The four bases of DNA are adenine, cytosine, guanine, and thymine, respectively abbreviated to A, C, G and T. The DNA is known to form a double helix, which is a structure consisting of two strands that are bound together in antiparallel form. Each strand is a nucleic acid sequence and because of the complementarity property shared between two nucleic acid sequences one know that they are bound in a particular manner – adenine bases complement thymine bases, and guanine bases complement cytosine bases. As a result we have four valid combinations of bases, called base pairs, and this structure of pairs of bases makes one strand enough to construct all the genetic information encoded in DNA [21].

2.2 Obtaining DNA sequences

DNA sequencing is the process of reading in the nucleotides of DNA, fragments at a time, and to then reassemble them in the exact same order that the nucleotides actually appear in the DNA molecules. The product of the DNA sequencing is most commonly digital sequences of the letters A, C, G and T that represent the nucleotide bases. Because these sequences potentially can be extremely long, it normal to use terms such as kilo base pairs (kbp), and mega base pairs (mbp), and even giga base pairs (gbp) to be able to respectively label one thousand, one million, and one billion bases at a time.

Scientific instruments known as DNA sequencers automate the process of sequencing. As a result of fierce competition between manufactures, there has been a large increase in the number of sequence outputs that sequencers are able to produce. The consequence of this is that the cost of sequencing the whole human genome has been reduced radically [23]. In January 2014 the first sequencer able to sequence a complete human genome at a cost lower that $ 1000 was launched. This has been an awaited goal in the genetics community ever since the '$ 1000 genome' catchphrase was recored back in December 2001, and was reclaimed to be the beginning of a new era in personalised medicine [24].

2.2.1 Delineation of genomic features

While the process of digitising living creatures is an impressive feat by itself, sequenced genomes serve little purpose if their underlying informational content and functionality is not unraveled. Projects such as The Encyclopedia of DNA Elements and Roadmap Epigenomic (ENCODE) is contributing to the ongoing work of delineating the functional elements encoded in the human genome. These pieces of empirical data are known as *genomic features*, and can be defined as any discrete region of the genome that have some biological function, meaning that it encodes for a defined biological product. One feature can for example be a contiguous segment that makes up a certain protein coding gene, while another could be a region consisting of chromatin structures [6].

To be able to do automated computational analysis on these features, e.g. to find correlations that could help answer questions about the cause of medical conditions, the information is stored in datasets known as *genomic tracks*.

2.3 Genomic tracks

It would have been impractical, and for many purposes useless, to directly work with the raw base pair letter output of the DNA sequencers when doing analysis. This is primarily because many analyses are more interested in the composition of the base pairs, and their meaning, rather than their respective letters. Hence we are dealing with genome-wide base pair positions, where the positions are based on the base pairs in internationally

accepted reference genomes [13] . The positions are referred to as *genomic coordinates*. Since the genomic coordinates are unambiguous and generically interpretable across genomes of the same species, they are suitable for pinpointing, or to use as a basis for annotating, genomic features. The division of DNA into chromosomes and scaffolds acts as a natural way of dividing the genome, so genomic coordinates are commonly relative to the start of a chromosome or scaffold. It is also worth noticing that genomic coordinates can either be zero-based or one-based, i.e. starts counting at zero or one. The significance of this will be addressed in Section 2.5.3.

Genomic coordinates are a simple abstraction of the genome that allow us to look at genomic features, and other genomic entities of interest, as generic *genomic elements* positioned on a line. An annotated collection of genomic elements is referred to a genomic annotation track, genomic track, or in context of genome analysis *track*. These tracks are genome-scale datasets with annotations that describe the elements, i.e. the underlying information that is stored in the raw DNA sequences that have been unraveled through empirical research. The genomic tracks allow us to do various kinds of analysis on genomic data. Not only may they be used to show a visual correlation of different types of information, they also provide a way to perform statistical analysis, which for example can answer statistical questions about whether there are base pair overlaps between two tracks, suggesting a relationship between the tracks.

The genomic tracks often contain positional information for each element, in form of the coordinates, but they can also include more properties. Some properties are required in order to support information related to different domains, technologies or experimental methods [13]. For instance, the BED format have an optional property field that is an RGB value that specifies the colour that should be used for that element when the track is presented graphically, that have nothing to do with the underlying genomic information, but is relevant to the displaying method. The fact that one might need a specific set of properties is one of the reasons for why there are several representational formats, and not only one. We will come back to the genomic track formats in Section 2.5.

2.4 The Genomic HyperBrowser

The Genomic HyperBrowser is a web-based system that provides various tools and functionality for analysis of genomic tracks. The system is built upon the Galaxy, which is an open source platform 'for performing *accessible*, *reproducible*, and *transparent* genomic science'. *Accessible* in the sense that life scientists with limited competence in usage of computational tools are able to use the web-service to do experimental analysis, *reproducible* so that results from performed experiments can be reproduced by another scientist; and *transparent* which means that scientist are able to share and communicate their experimental results [12].

Because the service is integrated with the Galaxy, all standard Galaxy *tools* are directly accessible from it. This opens up the possibility of using

these tools alongside with the functionally unique to the HyperBrowser, and even use The Genomic HyperBrowser such as a normal Galaxy instance [36].

Although the main installation is a publicly available web-service hosted by the University of Oslo[1], it is possible to install the HyperBrowser locally or on a web server of choice. The source code of the project is freely available online, through a SVN repository[2].

The HyperBrowser provides a sizeable library of genomic tracks, although users are allowed use custom tracks or other tracks obtained elsewhere [37].

2.4.1 Tools

A *tool* in the Genomic HyperBrowser can be an arbitrary piece of software, the only requirement is that it has a command-line interface. Some tools may for instance perform statistic operations on genomic tracks, such as the 'Count' tool, while others may simply present meta information in form of benchmarks for how a certain part of the system performs.

Developers can add their own tools to a HyperBrowser instance. When embedded as a tool in the HyperBrowser, the software gets a web interface that follows a common standard. The features enabled in the user interface, such as forms, buttons etc., are specified in an XML file, referred to as the tool configuration file. The config file also tells the HyperBrowser what the name of the tool is, how it is run, and specifies input and output.

2.4.2 Job history

A history gives a chronological overview of tools that either are running or have been run. Using Galaxy terminology, the history items are simply called 'jobs'. Jobs marked in green are done, jobs in yellow are ongoing, and jobs in red have stopped due to an error.

Histories can be converted into *workflows*. When a workflow is executed, all the jobs that it is comprised of are run in the same order as they were added to the original history. This makes it possible to reproduce a multistep analysis by a single click.

2.4.3 The abstract methodology of the HyperBrowser

When using the standard tools provided by Galaxy on a selected track, the users themselves have to come up with the appropriate statistical questions and the set of operations that must be applied to answer these. Typical legal questions are answered by following a similar multistep procedure. A special tool that comes with the Genomic HyperBrowser helps the user answering such questions by presenting a range of suitable forms of generic analyses, often making the analysation process faster and more convenient. The different types of analyses are divided into the categories 'Descriptive

[1] The main installation of HyperBrowser is located at: https://hyperbrowser.uio.no/hb/

[2] Guide on how to acquire the HyperBrowser source code: https://hyperbrowser.uio.no/hb/static/download.html

Statistics' and 'Hypothesis testing'. The first comprises statistics such as counts or average lengths, while the latter comprise hypothesis tests such as whether the segments of one track overlap more than expected with segments of another track.

The exact set of analyses that are possible to do on a given track, depends on what is known as the *track type*. The people behind the Genomic HyperBrowser have created an *abstract methodology* that defines the track type based on a special set of informational properties, i.e. properties that are directly related to the semantics of the genomic data. The Genomic HyperBrowser web service utilises this definition of track types to decide the analyses that are applicable to single, and pair of tracks [13].

2.4.3.1 Informational properties

As an effort to formalise the concept of genomic tracks, the Genomic HyperBrowser project have defined four core informational properties for genomic tracks – *gaps*, *lengths*, *values* and *interconnections*. Gaps refer to space in between elements, lengths to outstretched elements, i. e. segments, values to informational values associated with elements, and interconnections to connections with other elements that could be located elsewhere in the genome.

2.4.3.2 Track types

The type of a track is solely decided by the combination of informational properties. Four basic properties give a total of 16 distinct combinations. However, the combination that correlates to a track with no properties is of no interest and is excluded from the set of track types. This result in a total of 15 different track types, i.e. one track type for each valid combination of the informational properties. This binary notion of which properties a genomic track has, where a property either is included or excluded, can be represented as a *four-dimensional matrix* that can be found in Figure 2.1.

The 15 track types are divided into two groups, designated as the *basic track types* and the *extended track types*.

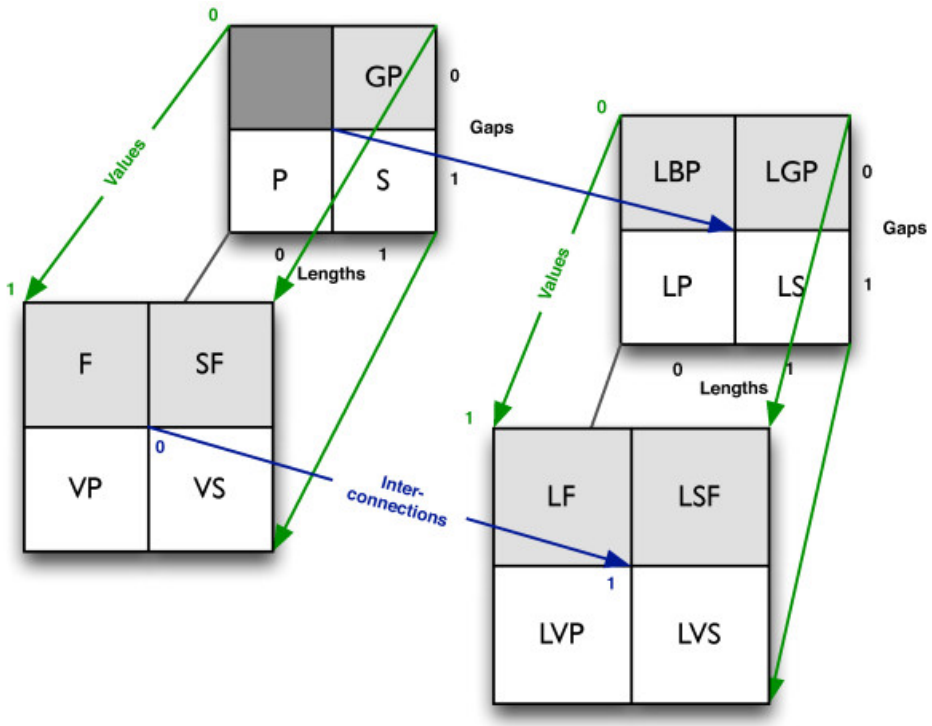
Basic track types

Points (P) Tracks that only have *gaps*. e.g. each genomic element represents a single base pair position.

Segments (S) Tracks that have *lengths* and *gaps*. e.g. each genomic element refers to a contiguous sequence.

Genome Partition (GP) Tracks that have *lengths*. e.g. the genomic elements represent a contiguous partitioning

Function (F): Tracks that only have *values*. e.g. each genomic element is a base pair that has an associated informational value that forms some continuous function.



Source: Sveinung Gundersen, Matúš Kalaš, Osman Abul et al. 'Identifying elemental genomic track types and representing them uniformly'. In: *BMC Bioinformatics* 12.1 (2011), p. 494

Figure 2.1: Four-dimensional matrix mapping the relations of the fifteen track types.

Valued Points (VP) Tracks with *gaps* and *values* i.e. **Points** with values.

Valued Segments (VS) Tracks with *lengths*, *gaps* and *values*. i.e. **Segments** with values.

Step Function (SF): Tracks with *lenghts* and *values*. i.e. **Genome Partition** with values.

Extended track types We get most of the less common *extended track types* by adding the interconnection core property to the basic track types. These are **Linked Points (LP)**, **Linked Valued Points (LVP)**, **Linked Segments (LS)**, **Linked Valued Segments (LVS)**, **Linked Genome Partition (LGP)**, **Linked Step Function (LSP)**, and **Linked Function (LF)**. The 15th and very last track type is where the track has only the interconnections property. This one is named **Linked Base Pairs (LBP)**, since base pairs in different places on the genome is linked or connected together by edges [13] [14].

Sparse and dense tracks A useful distinction related to track types that is commonly used in the HyperBrowser, or more specially in *GTrackCore*, is the one between sparse and dense tracks:

Sparse tracks: All tracks that have gaps, i.e. Points or Segments tracks, or variations of these such as Linked and/or Valued Points (VP/LP/LVP) and Linked and/or Valued Segments (VS/LS/LVS).

Dense tracks: All tracks without gaps, i.e. Function, Step Function, Genome Partition tracks or linked variations of these, or Linked Base Pairs. These tracks need to have *bounding regions* specified explicitly, which we will come back to and detail in Section 2.7

2.4.3.3 Valid operations on the various track types

By utilising the aforementioned methodology, one can say that the different operations that may be performed on a specific genomic track, depends on the type of the track. It makes no sense to do a count of points on a Segments track, or get the average length of segments on a Function track, simply because the information stored in the different track types differ. Some operations may also require a pair of tracks.

A few common operations that involve two Segments tracks:

intersect: finds elements of two genomic tracks that overlap.

overlap: gets the base pair overlap of two tracks.

subtract: finds the features that overlaps on two tracks. Then the overlapping sections are removed from the first input file, and the result is reported

2.5 Representation of genomic tracks

There are many ways to represent a genomic track digitally on a computer. The fashion in which the informational content of genomic tracks is laid out in a computer file is referred to as the track format. The track format decides not only how the information is encoded, but also which property fields, or *columns*, that may be included for each genomic element. There are mainly three ways the genomic tracks are represented; as textual data, binary data, and XML data.

2.5.1 Textual formats

The most basic representation of a genomic track can be seen of as a set of genomic elements, accompanied by positional information and possibly some other properties, stored as a human-readable text file. We say that this way of representing a genomic track as textual data is the most basic representation because it easily can be read, understood, and even created, by a human with knowledge only of molecular biology. At the same time a computer can parse it automatically. Textual formats are probably for these reasons the most common way of representing genomic tracks. Typically, textual formats have one element on each line, where some delimiter separates properties such as position coordinates and chromosome

numbers. The majority of the textual formats are tabular, i.e. the columns that correspond to the different properties are separated by tabs.

Currently the most used textual formats are BED, WIG (Wiggle Track Format), GFF (General Feature Format), and FASTA [42]. These are all attempts at defining a generic format, and to put an end to the tendency of creating ad hoc formats. Another attempt at creating a generic format has been done with the more recent GTrack format. We will in the next subsection give a brief description of the BED, before we immediately move on to the more feature-rich GTrack format.

2.5.1.1 BED

An example of a popular textual format is the tabular BED (Browser Extensible Data), which is divided into 12 fixed columns. Three of these columns must be defined: Chromosome number, start and end coordinates. The other 9 may be defined depending the information one wants stored. In conjunction with the track types defined by the HyperBrowser methodology the BED format supports Points, Segments, Valued Points, and Valued Segments. A part-of relationship is also supported, e.g. exons that are part of a gene, which is a special case of Linked Segments.

2.5.1.2 GTrack

GTrack is a new general-purpose tabular format that is an offspring of the HyperBrowser project, and it supports all the earlier described 15 track types. Many other tabular formats, e.g. BED and WIG, may easily be embedded in a GTrack file, by giving a syntactic description of the how the genome elements are formatted in a metadata section at the beginning of the file.

Metadata Lines prepended by hash signs (#) in a GTrack file are used for metadata, and are referred to as GTrack specification lines. A GTrack file typically has a few of these lines in the header. A single leading hash sign simply means that the line is a comment. Two hash signs are used for variables, which for instance could be boolean values that tells whether the elements are sorted or 1-indexed (see Section 2.5.3), or a string explicitly defining the track type. The line with three hashes, the column specification line, is especially important because it defines the content of all the columns in the file. This line is used to identify the four columns that correspond to the previously detailed core informational properties, which we will look into in the next paragraph. Lastly, a line with four hashes is used to specify a bounding region, which is mandatory for dense tracks [14].

Reserved columns GTrack have eight reserved column names. Four of these are associated with the four informational properties. Table 2.1 gives an overview of how these reserved columns are related to informational properties and track types. The correspondence between the informational properties and the columns that are used to represented them is perhaps

unclear; a *start* column marks the start coordinate of each element, and will indirectly define gaps between the elements. Start and end columns together will define lengths. The edges column also requires the presence of an id column for all the elements, making them referable for other elements.

Table 2.1: The reserved columns of the GTrack file format in relation to informational properties and track types.

Track type	genome	seqid	start	end	value	strand	id	edges
	N	*N*	*C*	*C*	*C*	*N*	*N*	*C*
P	?	!	*X*	.	.	?	?	.
S	?	!	*X*	*X*	.	?	?	.
GP	?	!	.	*X*	.	?	?	.
VP	?	!	*X*	.	*X*	?	?	.
VS	?	!	*X*	*X*	*X*	?	?	.
SF	?	!	.	*X*	*X*	?	?	.
F	?	!	.	.	*X*	?	?	.
LP	?	!	*X*	.	.	?	*X*	*X*
LS	?	!	*X*	*X*	.	?	*X*	*X*
LGP	?	!	.	*X*	.	?	*X*	*X*
LVP	?	!	*X*	.	*X*	?	*X*	*X*
LVS	?	!	*X*	*X*	*X*	?	*X*	*X*
LSF	?	!	.	*X*	*X*	?	*X*	*X*
LF	?	!	.	.	*X*	?	*X*	*X*
LBP	?	!	.	.	.	?	*X*	*X*

C Core reserved column (defines track type)
N Non-core reserved column (reserved, but does not define track type)
X Column is mandatory
? Column is optional
. Column is not allowed
! Property must be present, either as a column or in a bounding region specification

Source: Sveinung Gundersen, Matúš Kalaš, Osman Abul et al. 'Identifying elemental genomic track types and representing them uniformly'. In: *BMC Bioinformatics* 12.1 (2011), p. 494

2.5.1.3 Overlapping elements

Different elements may be overlapping. This means that the intervals represented by the start and end base-pair positions of two distinct elements, may cover the same coordinates. An example of a GTrack file with overlapping elements can been found in Figure 4.4

2.5.2 Binary formats

While textual genomic track formats have the advantage of being simple to parse by a computer, and may be humanly readable, they are not very compact. You need at least one byte for each character in the file, even for separator characters such as tab or space, to make the data appear in a form

```
##Track type: segments
###seqid        start        end
####genome=hg19
chr1         100         1000
chr2         500         1200
```

Figure 2.2: An example of a GTrack-file with two elements overlapping at base-pair positions $[500, 1000]$

that a human can interpret visually. By omitting the property of readability, the data can be stored in a manner that normally is more compact, because we won't have to use a whole byte at minimum to represent every character. It would for instance be a waste of space to use 10 characters to store 10 digit integers, when it is possible to represent integers of this magnitude only by using 4 bytes. Data in this form is called *binary data* and generally refer to data that typically can not be interpreted as human-readable text, but still is readable by a computer program.

Binary file formats are consisting of persisted binary data, and are more efficient to use than their textual counterparts. The reason why binary formats often are faster to work with is because they do not need to be parsed, and can be read into the main-memory directly. In addition they support incorporation of indexing schemes.

The binary formats are in many cases only used internally in the software, and not provided as public formats [13]. There are however a few examples publicly available binary formats. For example the BAM format, the binary compressed version of the Sequence Alignment/MAP format (SAM), or the bigBed and bigWig formats which are binary versions of BED and WIG.

2.5.3 Coordinate conventions

A convention regarding coordinates, which is decided by the representational format, is where to start counting from. Whether the coordinate identifies the base itself or the space in between two bases, and whether an coordinate interval is half-closed, i.e. if it has a exclusive start, and inclusive end.

Programmers often prefer 0-based coordinates, while biologists prefer the 1-based system. The reason why 0-based systems are preferred by computer scientists, and from a computational perspective, is partly because array structures, other collections, etc. in most programming languages start counting from zero. It also allow for effective calculation of the length of segments if the intervals are half closed, which is a very common operation. For example, we assume that we use coordinates that are half open and 0-based, if we then want to find the overlap of two tracks (x, y) and (z, v), we can then do the following:

$$overlap = min(y, v) - max(x, z)$$

On the other hand if the intervals where 1-based, we would have to do a somewhat more complicated computation:

$$overlap = min(y, v) - max(x, z) + 1$$

Although simple, this extra addition adds more complexity when done perhaps millions of times. It also make the code messier, which might be of at least the same importance as running time. Cleaner code makes it easier to guarantee that the code doesn't contain any bugs and that the analysis tools that it provides are reliable.

2.6 Technology

Genomic analysis often involves applying complex operations on very large genomic tracks. If you for instance have a Function track with values defined over all the 3,234.83 mbp of the human genome and a Segments track with an element for every defined gene, and you want to find the average value over all genes. This is a heavy operation that is working on two large data sets simultaneously to compute a result. For this kind of operation to complete in reasonable time the programming language, in which the operation is implemented in, has to perform well.[3]

Higher-level programming languages are generally preferable to lower level ones, since their high level abstractions increase productivity, and make it easier to argue that the tools produced by them are reliable. Their disadvantage is that they, at least in their 'vanilla' version, generally are much slower than lower level languages such as C. Immediately it might seem like there is a tradeoff between high performance and coding convenience, but this is not necessarily always the case. There are often extensions to higher level languages that make them perform much better in situations where performance is critical, such as in the previous example.

The upcoming sections will present technology important for the rest of the thesis. We will start with Python and give a listing of relevant topics and features. Secondly we will look at how NumPy fixes some of Python's performance issues. Then we will look at the storage library HDF5 (Hierarchical Database Format), and the very much pertinent PyTables, which is an abstraction layer on top of the HDF5. Lastly we will see how compression can be used to speed up data handling.

2.6.1 Python

Python is a widely used multi-paradigm programming language, that is emphasising code readability. The language is very expressive, and applications written in it usually consists of fewer lines than equivalent

[3]How well software written in some language performs, is strictly not a property of the language itself, but rather a property of the language implementation. However, even though an implementation of a high-level language could compile directly to C it would not mean that it would perform as well as hand-written and optimised C code. The only way to achieve this would be to have a really complex, and thus slow, compiler, : http://www.quora.com/Computer-Programming/Can-a-high-level-language-like-Python-be-compiled-thereby-making-it-as-fast-as-C/answer/Tikhon-Jelvis

applications in for instance Java or C++ [40]. Its long list of features, e.g. list comprehensions, list slicing, generators, and other syntactic sugar such as chained boolean comparison, makes writing code very convenient.

While all these features give programmers a lot of power at hand and enables them to write code faster, they tend to make the language perform worse. Lists in Python are for example much slower than arrays in optimised C, since Python is hiding the underlying workings of the machine behind high-level abstractions. Python is still popular among scientists and there numerous scientific tools written in the language, for instance the Genomic HyperBrowser and Biopython [4]. The reason for this is likely because it has for quite some years had a wide selection of extension libraries that are exclusively written in optimised C, thereby making the language highly usable in scientific applications where performance means a great deal.

In the HyperBrowser and in GTrackCore, Python is used to describe the high-level structure of the system, while the NumPy extension is used to handle the actual genomic tracks in-memory.

2.6.1.1 Code organisation

Modules In Python, a module is a file that can contains definitions of variables, methods and classes, or runnable statements. Statements located directly in the module scope, outside of any classes or methods, are executed the first time the module is imported. To keep a project tidy, functionality located in the same module should be logically related. A module may be imported from elsewhere through the `import` statement.

Packages A package is a directory that may contain modules and other sub-packages. Every directory with a `__init__.py` file is interpreted as a package by Python. This file is executed the first time something located inside of the package is imported. While often is kept empty, `__init__.py` can be used for preliminary setup of the package. e.g. It could for instance be imaginable that a common directory structure is required by all modules of some package. If the directory paths are created in the `__init__.py`, it is guaranteed that they exist when something inside of the package is called.

Classes The object-oriented paradigm is supported in Python, meaning that it language has a class construct that may be instantiated. Classes in Python have a special syntax for implementing certain operations; methods that start with double underscores are used two override standard functionality, and can be used in a unique way. For instance if a class has the `__getitem__` method defined it and `obj` is an instance of the class, then it is possible to access items by: '`obj[x]`'. Another important double underscore method is the `__init__` serves as a constructor that is run after the class has been instantiated.

In Java for instance, everything has to be in a class even when the only work of the class is as encapsulation of methods. In Python however, use of classes is discouraged when you are not in need of manipulating an instance.

In these cases methods should instead put directly in the modules, which results in fewer lines of code and a project structure that is more flat.

2.6.1.2 Pickling

Pickling is the processes of serialising a Python object, converting it into a stream of bytes. Through the `pickle` module of Python [31], a pickled object is allowed dumped directly to a file. The pickled object can then later be loaded and 'unpickled', a process which restores the original object. Hence, `pickle` can be used to serialise complete class instances. This allows all the object variables, methods, etc., that are associated with the class instance to be persisted and then to be restored in the exact same state later.

2.6.1.3 Dictionaries

A dictionary is a built-in data structure in Python, and is created by enclosing objects within two curly braces. In contrast to list or array structures, which are indexed by integers, dictionaries are indexed by strings, known as keys. Each key of a dictionary is associated with a value, and the keys can be used to do a lookup in the dictionary.

Shelf Is a persistent data structure that resembles a dictionary, and is located within the built-in `shelve` module. The values of the shelf can be anything that the `pickle` module supports, e.g. class instances.

2.6.1.4 File locking

File locking is a mechanism for limiting access to a file descriptor. Through the Python module `fcntl` [9], a file is locked by invoking `flock()`, with a file descriptor and the desired *lock operation* as input parameters, immediately after the file is opened. The lock operation has to be one of three constants that are used to specify the lock level:

LOCK_SH: To acquire a lock that may be *shared* between multiple processes. Typically used when there are multiple readers.

LOCK_EX: To acquire a lock that is *exclusive*, and prevents any other process access. Used when a single process need exclusive write access. Can only be acquired when no other processes are claiming.

LOCK_UN: To *unlock* the file descriptor.

2.6.1.5 Decorators

A decorator is utilising Python's functional capabilities: It is practically a method that takes in another methods, and then prepends or appends code to it. The preferred way to decorate a method in Python is by simply adding the '@' symbol, followed by the method that serves as a decorator.

A typical scenario where you can use a decorator is when you want to measure the run time of some method. An example of this can be found in Listing 2.1

Listing 2.1: Example of how decorators work. When a method is decorated with `timeit` the execution time is printed along with its standard output.

```
In [1]:def timeit(func):
           def inner(*args,**kwargs):
                      start = time.time()
                      result = func(*args,**kwargs)
                      print 'Time used:', time.time() - start
                      return result
           return inner

In [2]:@timeit
       def list_exp(list):
              return map(lambda x: x*x, list)

In [3]: list_exp([2,3,4])
Time used: 1.69277191162e-05
Out[3]: [4, 9, 16]
```

atexit.register This method, which may be used as a decorator, is an exit handler that registers a method to the `atexit` module [3]. A method registered to this module serves as a cleanup method that is run automatically when the Python interpreter terminates normally.

2.6.2 NumPy

NumPy [28] is an extension of Python, and a package for high performance array computation. At the core of the package is a `ndarray` object, which is a N-dimensional uniform array of elements. The package includes a library of functions that do vectorised mathematical operations on the `ndarray` data structure. The vectorised operations are implemented in C, and are significantly faster than operations that are using Python for-loops. They achieve this by grouping together element-wise operations, which is possible partly due to the fact that an `ndarray` is of homogeneous type and won't have to be repeatedly type checked, which would be the case for Python lists [43].

2.6.2.1 Memmaps

Memmap is a subclass of the `ndarray` that is used to create memory-mapped arrays stored in binary files. This makes it possible to handle very large `ndarray`, without having to have them entirely loaded into memory. If the mode of the array file is open in mode 'w', for write, changes can be done to any mapped portion, and are saved to disk automatically when the object is deleted or by manually by calling `flush()`. The `offset` parameter

determines where the mapping starts and is specified as a byte offset, which must be a multiple of the byte-size of the array's data type.

2.6.3 HDF5

HDF5 [41] is a technology suit built for management of large and complex data. First and foremost, the suite includes a portable binary file format, and provides a few interfaces that may be used to facilitate a wide range of features related to storage [17]. The API is written in C and implements the objects of what is known as the abstract data model of HDF5. The abstract data model is a conceptual model of data, data types and data organisation, and is independent of storage medium and programming environment. Objects of this abstract data model are mapped to the a storage model, which serves as a physical representation of the model that resides on some storage medium [19].

2.6.3.1 The HDF5 file

The HDF5 file is a container for organising a collection of *groups* and *datasets*. These objects respectively behave very similarly to directories and files of a file system. The HDF5 group is a structure entity used to hold multiple HDF5 objects, primarily other groups and datasets, and metadata that describes the its contents. Every HDF5 file has at least a root group ('/') that are serves as base of the internal file structures. The HDF5 *datasets* are on the other hand multidimensional array-like structures that contain the actual data, and are upon creation assigned properties such as name, dataspace, and datatype. The *dataspace* describes the layout or dimensionality, i.e. number of dimensions, while the *datatype* describes the actual contents of the dataset [10]. There are two main categories of datatypes:

Atomic datatypes String, integer, float, etc.

Compound datatypes A collection of atomic data types or arrays of such types, i.e. conceptually similar to structs in C.

2.6.3.2 Chunking

The HDF5 library makes it possible to specify how data should be stored on disk, how to address it, and how it should be kept in memory. Chunking is a technique where the raw dataset that is desired stored is split into smaller pieces, known as chunks, before these chunks then are written in arbitrary positions within the HDF5 file. The chunks are mapped by a B-tree that HDF5 keeps in memory.

If the chunks for instance are small enough to fit in the CPU cache it will greatly reduce the use of the memory bus, which in theory, if looked at in isolation, will mean a quite large optimisation [1]. However, many chunks will lead to large B-trees which in turn will cause more storage overhead, since it takes more time to access and maintain the B-tree. In practice it

is best to balance between I/O overhead related to managing B-trees, and actual data access time [10].

2.6.3.3 Filters

A feature of HDF5 is that it lets data pass through user-defined filters that are manipulating the data on its way to and from the disk. Filters require the data is chunked.

Filters can for instance be used with compression, as we will see in Section 2.6.5.2

2.6.4 PyTables

PyTables [2] is a Python package for managing large hierarchical datasets. PyTables is using the HDF5 library for I/O, and NumPy for managing data when it resides in-memory. The packages gives access to most of the C API of HDF5, albeit not all of it, and provides its own abstractions of HDF5 concepts like groups and datasets.

One of the unique features of PyTables is its ability to perform queries on tables, i.e. multidimensional structures of heterogeneous type. The queries are built up quite similarly to SQL queries of relational databases, and the engine that runs them are using NumExpr [27].

2.6.4.1 The PyTables file

The term *PyTables file* is used for HDF5 files created by the PyTables package. The only real distinction between this and regular HDF5 is that the `PYTABLES_FORMAT_VERSION` metadata attribute is set. The file extension of a PyTables file is up to the user to set, although '.h5' often is used since it is the default extension for HDF5.

When an PyTables file is opened, by invoking `tables.open_file()`, a `File` object is returned. A `File` object is associated with a single HDF5 file and offers various methods for editing the file's internal node structure, referred to the object tree of the file.

2.6.4.2 The object tree

The object tree of a PyTables file is imitating the HDF5 group and dataset tree structure stored on disk. PyTables has added an abstraction layer to deal with the nodes of the HDF5 tree, and for this the classes `Node`, `Group` and `Leaf` are provided.

By using a naming convention known as *natural naming* to name the nodes of the object tree, makes the object tree easily browsable To make the object tree easily browsable, the PyTables team has

Node Used to represent most kinds of nodes in the PyTables hierarchy. Since the class is abstract it can not be instantiated directly. All nodes of the object tree, except for file nodes which are handled through the `FileNode` module, are however descendants of this class.

Group Entity is used to manage groups of the HDF5 tree, and resembles a file system dictionary. Groups may be referred to by a path string, which gives the full path to the group. e.g. `list_nodes('/hg19/testcat/test')` gives a list of all the nodes that are hanging from the supplied path.

A group structure may also be traversed by using the `walk_groups` method, that returns each group of the object that are accessible from a branch parameter.

Leaf An abstract parent class of all the data container classes, i.e. different abstractions of the HDF5 datasets. `Leaf` nodes does not have any children.

Table A class representing a heterogeneous, chunked, HDF5 dataset consisting of records, i.e. a HDF5 dataset consisting of *compound* data types. The records are of fixed-length, and their contained fields must be decided before the `Table` is created. Records are more commonly designated as rows, represented by the `Rows` class, and `Tables` may grow arbitrary large in this this 'row dimension'. The supported number of rows is up to 2^{63} which should be enough for most genomic tracks.

A column of a `Table` is represented by the `Column` object, which may be accessed through the `__getitem__` method and modified through the `__setitem__` method. It is possible put indices on the `Columns`, which in theory should speed up queries.

Tables are created by using the create_table function of a PyTables object. The three required parameters of this method is the desired group path, the table name and an object that specifies the structure of the data. After the table has been created, rows may be added to it by extracting the `Table` node from the PyTables file object. An example of this process can be found in Listing 2.2.

Listing 2.2: An example of how a Table is created in a PyTables database file, and how a row is appended to the created table

```
h5_file = tables.open_file('testdb', 'w', title='test')
table = h5_file.create_table(h5_file.root, 'test', {'chr':
    tables.StringCol(10), 'start': tables.Int32Col(), 'test')

row = table.row
row['chr'] = 'chr1'
row['start'] = 2000
row.append()

table.flush(); table.close()
```

Array A class used for datasets that are homogenous, i.e. a HDF5 dataset consisting of *atomic* data types. There are a few variations of it, and which to use depends on whether the arrays should be *extendable*, i.e. have one of its dimensions enlarged such as a table, or *compressible*. `EArray` supports both of these features, the `CArray`, or 'chunked' array, is only

supporting the latter, while the basic `Array` class supports neither. All these subclasses may be initialised with a NumPy `ndarray` by setting the named argument `obj`.

2.6.4.3 FileNode module

This module provides an interface for adding regular files to the object tree of a PyTables database file. The files may then be read and written as any ordinary Python file. See Listing 2.3 for an example of how the module works.

Listing 2.3: An example of how a new read-write file is created in a PyTables database file by using `FileNode`

```
from tables.nodes import filenode
h5_file = tables.open_file('testdb', 'w', title='test')
node_file = filenode.new_node(h5_file, where='/', name='test node')
node_file.write('test') # can be used as any other file in Python
```

2.6.4.4 Setting chunk size

PyTables tries to reduce disk and memory usage to an absolute minimum, which involves to carefully set the chunksize. The safest way of setting the chunksize in PyTables is to provide the `expectedrows` parameter upon creation of `Tables` and `Earrays`. This parameter determines a sensible chunk size based on the expected number of rows, which is an estimation of how many rows that will be stored in the dataset.

Chunksize may also be set manually, by providing the `chunkshape` parameter. This variable specifies the number of rows that should be stored in a single chunk. If data, for example, most often is accessed by fetching 10000 rows at a time, e.g. through a column-slicing. it would be sensible to use a chunkshape as close to 10000 as possible. How close it can be set depends on the size of the datatype of all the columns combined. It is important to note that setting the chunkshape can have negative consequences if it is set to large, since HDF5 relies on having chunks that fits in the CPU cache. If the chunks exceed the cache size, HDF5 would have to use the main memory when they are loaded from disk, which would lead to a serious decrease in performance.

Note that the chunkshape has to be of rank 1, i.e. single-dimensional, since it only specifies the number of rows in a single chunk.

2.6.5 Compression

Compression involves to encode information so that the result end up using fewer bits that the original representation.

When working on very large datasets, compression is a possible way of optimising disk usage. Because use of the CPU comes at much lower cost that external disk access it should in some cases, when the files are sufficiently large and the compression ratio is decent, be faster to the

compress data before it is transferred to the disk. Less data would then need to be transferred back into memory, when the dataset is requested.

2.6.5.1 Compression ratio

The compression ratio is a metric used to quantify the reduction in compression of a given dataset [45]. It can be calculated by the following formula:

$$\text{compression ratio} = \frac{\text{compressed data}}{\text{uncompressed data}}$$

e. g. if the uncompressed data takes up 10000 bytes and the compressed data is 5000 bytes, then the compression ration would be $\frac{5000}{10000} = 0.5$

2.6.5.2 Blosc

Blosc is a lossless compressor that is optimised for speed rather than high compression ratios [1]. It is created by the founder of the PyTables project, Francesc Alted, and is the recommended way to compress datasets using PyTables. As compressors natively are supported by the underlying HDF5 format, by letting the data pass through a filter that serves as a chuck compressor/decompressor, compression and decompression happens in the in the cache of the CPU, making it occur very efficiently.

Blocking technique In order to cope with CPU starvation, which basically means that the CPU is waiting for slower components preventing it from realising its maximum processing capability, the Bloscs compressor/filter is using something called the *blocking technique*. This technique is leveraging three other tricks: Reuse of a dataset several times by caching it (temporal locality), make sure that the dataset is accessed sequentially from memory (spatial locality) and predict when a certain chunks will be used and transfer it to the CPU cache beforehand (prefetching) [1].

2.7 GTrackCore

The code that constitutes the GTrackCore package [15] started out as the tightly integrated module that handles storage and retrieval of binary genomic tracks in the Genomic HyperBrowser. GTrackCore is therefore naturally written in *Python*, as the rest of the HyperBrowser, and is using *Numpy memmaps* as its internal binary data format. This extraction, of what is perhaps the core functionality of the HyperBrowser, originates in an idea that the original developers had about making it possible to incorporate the powerful *preprocessor* that comes with the HyperBrowser, that handles most textual filetypes including their own GTrack format, in an envisioned command-line based analysis toolset.

At some point, it is also planned to have the integrated data model of the Genomic HyperBrowser replaced by GTrackCore. Having this functionality in a loosely coupled package rather than in a tightly integrated module,

makes it easier to experiment with it. It would also be advantageous to have this core functionality shared between the HyperBrowser and the proposed command-line based tool, since changes in GTrackCore then would affect both.

2.7.1 Bounding regions

Bounding regions are important to data retrieval in GTrackCore. The bounding regions of a given genomic track are the collection of all the known or defined genomic region, and can be said to constitute the domain of the track. Any retrieval has to take place within them. A thing to notice for *sparse* tracks is that absence of elements also is considered to be information, meaning that a bounding region is a region where lack of data means something. Parts of the genome that has not been investigated should be left out of the bounding regions. [14]

In GTrackCore, the bounding regions are stored in a shelf, and contain the identifier of the bounding region along with positional information about the area of the track which they are encompassing. Bounding regions are by default set to be all chromosomes, meaning that every base-pair of the genome is encompassed by a bounding region (which strictly speaking, according to the definition above, signifies that everything of the genome has been investigated). They may, however, be defined explicitly, and this is required for *dense* tracks (see Section 2.4.3.2).

2.7.2 The data format of GTrackCore

For its binary format, GTrackCore is using NumPy memmaps. Having one n-dimensional memmap for each column of the track. For example, a 'start' column could be represented as a one dimensional array of datatype `numpy.int32`, while a column that specifies a chromosome could be a 5 bytes string array. To extract the whole genomic elements, the same index has to be used in all the different column memmaps. For example, if a track only has a 'start' and an 'end' column the content of the 5th genomic element is extracted by invoking `__getitem__` of each column: `s = start[4]; e = end[4]`

2.7.2.1 LeftIndex and rightIndex

A track in GTrackCore is conceptually split into regions of preset size. These regions are known as *bins*. Two index structures, the `leftIndex` and `rightIndex`, use the concept of bins to accelerate lookup of regions, which are defined by a start and end base-pair position, in the possibly large column memmaps that constitutes a track. For these the queries regions are used directly to

The index structures keep track of where the first genomic element, i.e. that has the lowest start or end base-pair position, of any defined bin is located in the column memmaps. The `leftIndex` considers start base-pair positions while the `rightIndex` considers the end base-pair positions. For example, if the bin size is 100000, and we have a

memmap with 'starts' $[90000, 95000, 96000, 110000]$ and a memmap with 'ends' $[95000, 106000, 205000, 215000]$, then the `leftIndex` $= [0, 3]$ and the `rightIndex` $= [0, 1, 2]$.

Since the indexes are considering the 'start' and 'end' columns they are not used for tracks that have a 'dense representation', i.e. tracks that does not have 'start' or 'end' columns.

2.7.2.2 SmartMemmaps

The SmartMemmap is basically a wrapper for NumPy memmaps, and gives a view over a specific column index range. The wrapper's responsibility is to cache memmeps, so that new ones does not have be created.

Chapter 3

Development Practices

This chapter will outline the process and the development practices that were utilised along the way. We will start to present how we got acquainted with the code base. We will then move on to describe how the process of incorporating PyTables in GTrackCore was conducted, which involved creating an initial prototype that was later transformed into the final implementation. We will detail how structured testing positively affects software development, and how GTrackCore in advance was facilitated with an adequate test environment. Further, we will look at various methods for measuring performance, which includes simple *average of N* benchmark testing and detailed profiling. The chapter will conclude with two development methods that were used in order to make the collaboration work out.

3.1 Getting acquainted with the code base

Even though the GTrackCore packages only constitutes a small part of the Genomic HyperBrowser, it is quite complex. For new developers to be able to contribute to it, they need a basic understanding of how the system is designed. To aimlessly start implementing features is not going to work, and one needs some sort of a plan on how to proceed. This plan usually starts with finding a way of getting acquainted with the code base, which typically involves to read up on code documentation, if there are any.

The GTrackCore package has been developed in a research setting, where functionality is developed on problems that have not been fully understood from the start. In that kind of a setting, where code is produced along with the problem area continuously expanding, it is hard for the developers to find time to write good documentation. Due to this, the original developers made a decision to limit the time spent on writing documentation, as they would likely need to rewrite it later. As a result, large portions of the code base lacks documentation. To make up for this, the system has been implicitly documented through descriptive naming of classes and variables. This makes it manageable for new developers to work with the codebase despite its lack of documentation.

3.1.1 Assumptive documentation

The chosen approach to get acquainted with the code base was to read the code and make assumptions about how it worked at the method level. Following the flow a *preprocessing job*, we engaged in what can be described as 'assumptive' documentation, where docstrings expressed different levels of certainty about the inner working of different parts of the code base. A typical doctoring could look like the following:

```
def _allGESourceManagers(self, trackNameList, allowOverlaps):
        """ ?!
        Generator that return all the GESourceManagers. Return one
            GESourceManager that handle overlaps, and one where
            overlaps are merged
        """
```

In this example the leading '?!' mark denotes that the purpose of the method likely has been understood, but that it needs to be confirmed. Following up on this syntax, '??' is used to express that one is clueless as to how a method works. In a review session, when walking through the code with a developer with deeper knowledge of the code base, one can easily search for all occurrences of these marks in the source code and correct possible misconceptions. The docstrings can then be made final by removing the marks.

3.2 Implementation strategy

As the scope of the work had to be limited, we could not redesign every part of GTrackCore to make it fit perfectly with PyTables. Instead there has been taken a retroactive approach where we have tried to replace only the modules directly relevant to storage and retrieval. Minimal changes were thus made to the rest of the code base, which allowed us to fully concentrate on the main point of the assignment: To find out whether PyTables and the underlying HDF5 format was a good replacement. This means, for example, that the modules responsible for parsing textual genomic tracks, which we will look at Section 4.2.2.2, were kept untouched. However, it is conceivable that these could have been tailored to fit the storage method so that it would perform better.

From the outside, most of the modules and classes that were changed are working as before, i.e. the same public methods are provided.

Embedding PyTables into GTrackCore is a comprehensive task, involving change of how genomic track data is both stored and retrieved. The fact that these two separate modules of the system are directly dependent on each other makes the replacement process more complicated. To be able to conclude that the *preprocessor*[1] is working correctly one has to be able to retrieve the persistent data created by it from the retrieval module of the system. The dependency goes the other way as well, as a prerequisite for

[1] Which is where the storage of genomic data happens.

knowing that that any retrieved data is correct is that the preprocessor has stored it correctly. In other words, development of two different parts of the system had to done in parallel.

It would be naive to try make the preprocessing of all track types fully working with the first attempt. Hence, the strategy was to build the new PyTables version of GTrackCore incrementally, to gradually support more and more of the old functionality.

3.2.1 Creating an initial prototype

While PyTables was the proposed solution to the various problems of the ad-hoc memmap-based model and binary-format, it was not absolutely certain that it was an appropriate replacement. It would have been unwise to spend too much time trying to learn all the requirements of the implementation, i.e. how the system is supposed to behave in different situations on different kinds of input, and to then figure out the new design that could satisfy these requirements. If the assumed requirements were wrong, or if PyTables proved to be deficient, a lot of work would have been in vain. Thus, to minimise the risk of wasting time, we started out by trying to create a proof-of-concept *prototype* that only supported what was classified as the most basic functionality, i.e. only parts of the presumed software requirements were implemented, in order to learn the actual requirements [8]. The refined learning process that the prototyping proved to be, helped us immerse ourselves in the biological domain, the development environment, the Python programming language and the PyTables package.

The most basic requirement that was clearly understood from the very beginning, was that GTrackCore had to support simple Segments (S) tracks, with only start and end coordinates (see Section 2.4.3.2). Hence, the initial prototype was built to at least support this single track type. Complementary retrieval functionality, that made the same track type acquirable, was first added when it was confirmed that the preprocessed data was stored correctly, and that PyTables was working according to expectations.

3.2.1.1 Manually inspecting the preprocessed data

While the preprocessor relies on having the retrieval module up and running to be thoroughly tested, it is still possible to obtain a sense of whether it works by inspecting that the data fed to is being transformed into binary data. This kind of exploratory testing was done by using a HDF5 data browser named HDFView [20], which gave a visualisation of how the data sets were stored in the PyTables/HDF5 file, quite similar to how a file manager provides a view of a file system.

3.2.2 Evolving the prototype into the final implementation

During the creation of the prototype, the confirmed basic requirements, for instance that the system had to support preprocessing and retrieval of single-file Segments track, were implemented in a rigorous fashion. Hence

the initial code did not have to be thrown away when the prototype was working as envisaged. In other words, an evolutionary prototype [8], that could be evolved into a full-fledged implementation, had been built.

Further support for more complicated system requirements was implemented, e.g. support for preprocessing of Linked Valued Segments (LVS) tracks contained in multiple files (implementation details will be presented in Chapter 4). When it came to this, i.e. that the prototype was to be evolved into the final implementation, the aforementioned manual method of ensuring that components of the system were working correctly had to be discontinued, as the different track types and their peculiarities would have made the method impractical and too time-consuming. Instead, the existing test environment that came with GTrackCore was configured to work with the new PyTables version, thereby having the testing done in a more automated manner.

3.3 Eliminating bugs through structured testing

Debugging by solely inspecting that the output corresponds with the input is insufficient when implementing support for the many functional requirements of a complex system such as GTrackCore. A common practice in software development is to write tests before code. Meaning that the tests define the features. A feature expressed as a test is a very precise definition of how the method that implements the feature should work. In order to even write the test, the developer needs a very clear understanding of the requirements. Every scenario that might occur, from the trivial ones to the more complicated and composite, has to be explicitly defined. Although often being a tedious process, it saves development time. If writing a test is hard, it probably means lack of understanding of how the feature is supposed to work, forcing the developer to seek out this information.

3.3.1 Tests supplied with GTrackCore

The Genomic HyperBrowser is a system used by life science researchers to do analyses on a day-to-day basis. To be able to guarantee that the results produced by these analyses are correct, it is of utmost importance that the system as a whole is practically bug free. Consequently, a large number of tests are supplied with the system, including tests for the different components of the storage and retrieval functionality at its core. Hence, when GTrackCore was extracted from the HyperBrowser, these tests naturally came with it.

Soon, after the initial prototype had proven that the PyTables package was worth looking into, the supplied test environment was set up. This was a fairly simple process as the external behaviour of any classes or methods had not been changed.

In the GTrackCore code base, tests are located in the `test` package in the root of the project, which contains a collection of unit and integration tests. The tests are written by using the *nose* test automation framework [29].

Listing 3.1: Example test case for a basic Segments track. The first few arguments, line 1 to 6, define a parser and a textual genomic track that will be input to the preprocessor. Line 8 and 9 specifies two genomic elements identical to the ones residing in the textual data, and are used in various assertions. The remaining arguments are containing other information about the track, such as valid bounding regions, parameters and data types.

```
1  Case(GtrackGenomeElementSource,
2       'TestGenome',
3       [],
4       ['\t'.join(['chrM','100','165']),
5       '\t'.join(['chrM','200','299']),
6       '.gtrack',
7       ['example test case','gtrack'],
8       [GenomeElement('TestGenome', 'chrM', start=100, end=165),
9       GenomeElement('TestGenome', 'chrM', start=200, end=299),
10      [BoundingRegionTuple(region=GenomeRegion('TestGenome',
            'chrM', start=100, end=299), elCount=2)],
11      GtrackGenomeElementSource,
12      ['start', 'end'],
13      'float64',
14      1,
15      'float64',
16      1)
```

After installing nose, the tests are run by running the command `nosetests` in the test directory.

The effect of structured testing on the development will be discussed in Chapter 6.

3.3.1.1 Integration test cases

The integration tests of GTrackCore, i.e. tests that validate that the preprocessor and retriever combined are behaving as excepted, are run on 50 different *test cases*. These test cases are differently formatted textual genomic tracks, that together constitutes all track types. The expected behaviour is expressed as generic *assertions*, which are predicates that should always be true. If one of the assertions of a test evaluate to false, the test *fails*, meaning that the entity that it covers does not work as expected. The assertions are made generic so they can work with a range of different test cases.

Using the test case in Listing 3.1 as an example, a typical assertion could for instance be one where the hard coded `GenomeElements`, i.e. genomic elements, are asserted to be equal to the track elements output by the retrieval module (after first being exported to the binary format), Another could for assert that all elements are enclosed by the bounding region. If these two assertions alone made up a given test, and the test was run with the mentioned test case, the test would have *passed*. If a test passes it means

that the entity that it covers should be 'working' – at least according to the developer that wrote the tests.

3.4 Performance measurement

The first objective when doing significant changes to the data model that constitutes GTrackCore, is to ensure that it works in the same way as the old. The second is to determine whether the changes are beneficial to the performance of the system. For PyTables to be considered a good replacement of the ad-hoc memmap-based model it is not enough that it is bug-free, it also needs to be efficient. Not necessarily outperform the old version of GTrackCore by far, but at least perform on the same scale. GTrackCore definitely has high performance as a requirement, since one of its main tasks is to preprocess textual tracks to binary data quickly, so that analysis can be commenced as soon as possible. The other main task is to efficiently fetch binary data so that the analysis tools not are kept starving.

3.4.1 Benchmarking

A simple way to evaluate how a certain method performs, is to measure the 'real' wall-clock time it takes for it to finish, i.e. the actual time elapsed between invocation and termination of the method. This is often referred to as simple *benchmarking*. Benchmarking can be done by registering the time right before the method is entered and right after the same method exits. The elapsed time can then be found by subtracting the start time from the stop time. To make the result more accurate, the time may be averaged over a set number of runs.

How timing may be carried out using the `timeit` decorator was presented in Section 2.6.1.5. If, for instance, one wishes to time preprocessing of a certain track with GTrackCore, one can simply apply this timer to the method that initiates the preprocessing job. The number gives a measure for how well the system performs, and can be compared with an equivalent run of the memmap version of the system. If the number is smaller for the new version, it means that the system is faster. If, on the other hand, the number is larger, it means that PyTables performs poorer.

3.4.2 Basing optimisations on profiles

While the previously mentioned method to measure performance, i.e. benchmarking by taking the average run time of N runs, gives a single number that indicates how well various parts of your system performs, it will not give an answer to the question of *why* these parts for instance perform worse than excepted, or bad in comparison with another version of the system (e.g. PyTables version vs. memmap version of GTrackCore). Although it is possible to use intuition to answer this question, and base the decision of what parts of the code to optimise on this, it is a good chance that you waste your time on improving the speed of non-critical parts of the system. A detailed *profile* is a much more reliable source to

base this decision on, and gives a precise answer to the question of *why* parts of a system perform as they do. A profile prints statistics about how the different methods of the system performs individually, and is usually the preferred technique to find the exact components that are causing a given performance problem.

3.4.2.1 cProfile

The profiler that has been used during the development of GTrackCore and in interpretation of results is *cProfile* [7]. cProfile provides monitoring of events such as function calls, function returns and exception events. Precise timings (wall clock time) are performed for every event, making the results from the profiles reliable. However, some performance overhead incurs when profiling by cProfile, although this overhead typically is less than for other Python profilers. The overhead mainly concerns the time it takes before the profiler gets the time after it has been requested, and that the timings are limited by the underlying *clock tick rate*. The latter means that methods that take less that a clock tick not will show up correct in the profiles. A profile method entry of 0.00000 seconds can for example, in reality, mean that each run of the method takes 0.0000005 seconds.

Textual output of a profile run of the preprocessor, by use of cProfile, can be found in Figure 3.1. As seen from this example, there are various kinds of measures that give information about how the different methods involved in the run are performing:

ncall: The number of times the method has been called.

tottime: The total time spend in the method, *excluding* the time spent in all sub-methods (local time).

percall: The quotient of the **tottime** divided by **ncalls**.

cumtime: The total, or cumulative, time spend in the method, *including* the time spent in all sub-methods.

percall$_2$: The quotient of **cumtime** divided by **ncalls**.

The cumtime and tottime are the most interesting measures, and profiling runs are typically sorted on one of these. In the example, the method call stack is sorted on cumtime in descending order. For example, comparing cumtime of a method that is found in both the PyTables and memmap versions of GTrackCorecan tell us which of them is more performant. If, for instance, the PyTables version spends much more time in a method, it is a good indication that there might be something to optimise.

The profile runs that has been used to test the PyTables-based preprocessor outputs, in addition to the previously presented text profile, a binary profile 'dump' that is more detailed. This file can, for example, be opened and analysed with the GUI utility *RunSnakeRun* [35]. The HyperBrowser tool that we have created, with the purpose of making profiles as well as benchmarks reproducible, will be explained in the upcoming section.

```
Dumped profile that can be used with "Run snake run" (id=1405374064.73)
--- Profile ---
         1215387488 function calls (1215338852 primitive calls) in 2645.642 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.003    0.003 2645.647 2645.647 .../PreProcessTracksJob.py:45(process)
        2    0.000    0.000 2624.404 1312.202 .../PreProcessGeSourceJob.py:16(process)
        2   35.972   17.986 2624.404 1312.202 .../PreProcessGeSourceJob.py:22(_createPreProcFiles)
        6    0.000    0.000 1351.649  225.275 .../GESourceManager.py:151(getNumElements)
      144  198.868    1.381 1351.648    9.386 .../GESourceManager.py:49(_calcStatisticsInExtraPass)
 10219846   45.607    0.000 1063.806    0.000 .../GEBoundingRegionElementCounter.py:21(next)
 10464484   16.170    0.000 1012.063    0.000 .../GEDependentAttributesHolder.py:18(next)
 10464485   44.589    0.000  995.894    0.000 .../GenomeElementSource.py:122(next)
 10219846   70.984    0.000  984.516    0.000 .../GEOverlapClusterer.py:45(next)
 10596261  255.008    0.000  931.295    0.000 .../BedGenomeElementSource.py:30(_next)
 10464486   16.957    0.000  687.494    0.000 .../TrackGenomeElementSource.py:122(next)
 10464484   27.018    0.000  670.537    0.000 .../TrackGenomeElementSource.py:116(_wrappedTrackElsGenerator)
 10464482  232.701    0.000  599.394    0.000 .../GenomeElement.py:6(createGeFromTrackEl)
 31525228   65.021    0.000  197.633    0.000 .../GenomeInfo.py:249(isValidChr)
 10342163   16.162    0.000  196.237    0.000 .../OutputManager.py:80(writeElement)
 10342163  146.590    0.000  180.075    0.000 .../OutputManager.py:53(_add_ge_dict_as_row)
  ...
```

Figure 3.1: An excerpt of an example profile of the preprocessor of GTrackCore.

3.4.3 HyperBrowser tools for performance measurement

In order to have a common platform to run different performance tests on, and to make the results from them reproducible, a separate instance of the HyperBrowser was set up at https://hyperbrowser.uio.no/gtrackcore/. Here we developed two different test tools, namely the 'Operations performance tool' and 'Preprocessor performance tool'. The latter is depicted in Figure 3.2 on the facing page.

The tools let the users compare the two versions of GTrackCore, to see which of them performs better in different situations. The two supported test types are 'detailed profiling' or 'average of N runs'. The tests can either be run on a few *selected test tracks*, or on an arbitrary genomic track from the HyperBrowser's built-in track collection. The selected tracks are believed to represent most of the interesting flow paths that the system might take during execution. Because some tracks are more interesting to preprocess than to perform operations on, and the other way around, the selected tracks differ in the two tools.

There are a few more variables we have made it possible to test. For the PyTables version it is possible to switch on different levels of Blosc compression (see Section 2.6.5.2). We have additionally made it possible to switch on creation of arrays. What this means will be detailed in Section 4.3.2.3.

The Preprocessor performance tool is made available in Appendix A.

3.4.3.1 The selected test tracks

Bendability:

> **Based on:** `DNA structure:Bendability` (hg18)
> **Note:** A Function (F) track of 689 702 695 genomic elements. Has a single 'val' column. Has been reduced in size compared to the original, which covered every base-pair of the human genome.

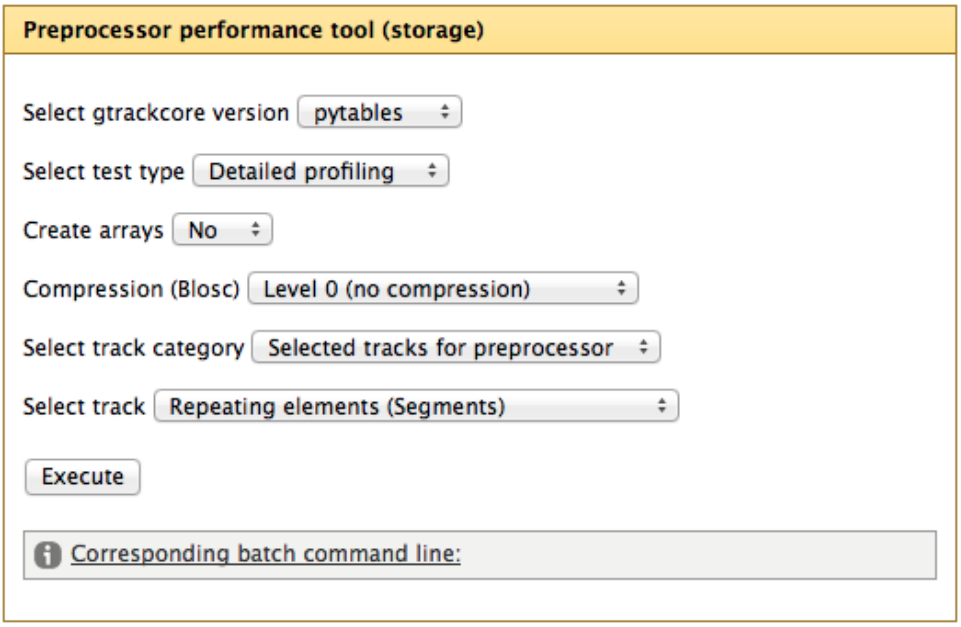


Figure 3.2: The Preprocessor performance tool. Used to make the results reproducible.

Repeating elements:

Based on: `Sequence:Repeating elements` (hg19)
Note: A Segments (S) track of 5 232 241 genomic elements. Has 6 columns.

Repeating elements$_2$:

Based on: `Sequence:Repeating elements` (hg19)
Note: A Segments (S) track of 5 232 241 genomic elements. Has 12 columns.

Interconnections:

Based on: `DNA structure:Hi-C:Inter- and intrachromosomal:hESC:hESC-1M` (hg19)
Note: A Linked Genome Partition (LGP) track of 3 076 genomic elements. Has 4 columns. Each element has a lot of edges.

Parkinson's:

Based on: `Phenotype and disease associations:GWAS:NHGRI GWAS Catalog:Parkinson's disease` (hg19)
Note: A Points (P) track of 67 genomic elements. Has 16 columns.

Sequence:

Based on: `Sequence:DNA` (hg19)
Note: A Function (F) track of 182 798 genomic elements. Has a single 'val column. Each row is a nucleotide sequence, where nucleotide is represented by a letter.

Genes:

Based on: `Genes and gene subsets:Genes:Ensembl` (hg19)
Note: A Segments (S) track of 182 798 genomic elements. Has 12 columns.

3.5 Methods of collaboration

When two individuals are working in tandem on the same Master's project, it may open for more frequent discussions and rapid exchange of ideas. Activities that single students increasingly are forced to engage in on their own initiative. For a close collaboration to succeed, it's a good idea to stick to recognised development methods and techniques.

The collaboration activities we have participated in includes pair programming and usage of a code version control system. These will be further elaborated in the upcoming sections.

3.5.1 Pair programming

Pair programming is a development technique where two programmers work together on the same computer to produce code. One person is responsible for the actual typing, and is often called the *driver*. The other person is often named the *observer*, and his or hers task is to prevent the driver from making mistakes, to come up with alternative strategies and to help detect design flaws. Many mistakes are in other words caught as soon they are typed instead of in the review session. Two sets of eyes also make it easier to avoid pitfalls that will lead to a bad design.

Research has shown that pair programming leads to higher quality code, faster completion of features, as well as happier programmers [47]. Another benefit is that the participants learn significantly more about the system and about software development in general. An experiment performed at the University of Utah in 1999 [5] showed that a group of students working in pairs completed the selected assignments 40% to 50% faster than a group of individuals. Additionally, the group of pairs passed a higher percentage of the test cases with 15% fewer defects, and they implemented the same functionality in fewer lines, which was believed to indicate that the pairs code featured better designs.

3.5.2 Version control using Git

It is crucial that applied changes are kept intact when several people are working on the same source files, even when they are working on the same lines of code. It is also important that changes are easily reverted, so that earlier versions, or revisions, of the codebase are accessible and even recoverable. A version control system, or VCS, maintains these two requirements, and commonly comes with a handful of other features that eases the development process.

When developing GTrackCore we used the decentralised VCS Git [11]. Decentralised means that each contributor has their own full-fledged local repository that they are working on, so that operations such as commits,

showing the history and reverting changes are fast [46]. In addition to providing common VCS operations such as commit, push and pull, Git also has some more advances features. These include the debugging command `git bisect` which utilises binary search to efficiently find a broken commit.

3.5.2.1 GitHub

GitHub is web-based code hosting service for software development projects that are using Git. To be able to put code up on Github, and make it publicly available to others, one has to create a public GitHub repository. By pushing code to this remote location, anyone interested are able to follow the development and come with suggestions, although only people marked as *contributors* can make code changes. Typically, if one wants to do independent development on an existing repository, one has to create a *fork* of it, which basically means to make an exact copy of the original repository.

By default, a repository has a single branch called the *master* branch, but more may be created. In our fork of the GTrackCore repository we kept the master branch intact with code as it existed prior to the replacement. All code related to PyTables was done in a branch called *pytables*. This made it convenient to switch between the two, for instance when we needed to find out what was the expected behaviour when our code was not working.

See Appendix B for the URL of our GTrackCore repository fork.

Chapter 4

Implementation of PyTables-based Preprocessing in GTrackCore

This chapter will give a detailed description of the preprocessor of the GTrackCore package, and how it has been adapted to use PyTables. It will thus cover all parts of the code base that have been involved in the replacement of the old binary data format, which is based on NumPy memmaps.

Detailed profiles, based on the 'selected test tracks' (see Section 3.4.3.1), will be referred to when it is found necessary to justify implementation choices, and when other possible solutions are discussed. These are all made available in Appendix A.

4.1 Distinguishing different versions of GTrackCore

Before the work of incorporating PyTables into the data model of GTrackCore began, there was only a single version of GTrackCore [15]. This project has led to the rise of another version which we during the development had the habit of calling 'the PyTables version of GTrackCore' [16].

To clarify the distinction between the different versions of GTrackCore in a concise manner, we will label the memmaps version of GTrackCore *the old GTrackCore* and the new, PyTables version, will be labeled as *the new GTrackCore*. If the words 'new' or 'old' are omitted when mentioning GTrackCore, it either means that the implementation is the same for both versions or that the package is described at a higher level.

4.2 Overview

The GTrackCore package has two main points of entry. One of them lets the user *preprocess* textual genomic tracks, and the other lets the user *retrieve* preprocessed data. The module responsible for the preprocessing converts

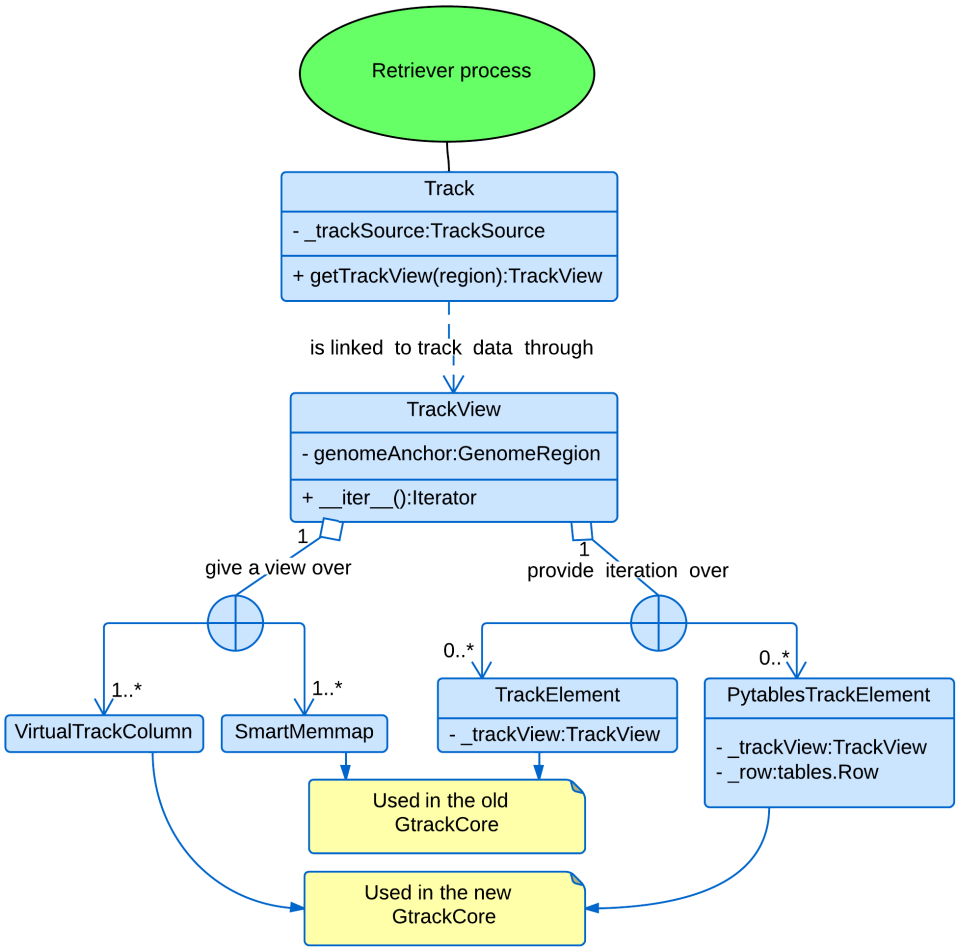


Figure 4.1: Composition of classes involved in a process' retrieval of track data.

textual genomic tracks into two distinct binary representations. The second module responsible for the retrieval provides an interface that allows for manipulation of the binary tracks created by the preprocessor.

As this project is a two-folded one, and this part deals with the preprocessing, i.e. the storage component of GTrackCore, we will not go into implementation details about the retrieval part of the system, even though much work has gone into that part of the system as well. Details about retrieval of track data can be found in the Master's thesis that makes up the second half of this work [38].

We will, however, soon see that track preprocessing depends on the module that retrieves data. Because of this we will start by giving a brief overview of the retrieval process. We will then go on to give a more thorough description of the track preprocessor.

4.2.1 Retrieve binary track data

In GTrackCore, preprocessed tracks are retrieved by instantiating the `Track` class. As the track name serves as a unique identifier for tracks, `Track` objects are initialised with the track name represented as a list of strings. All further interaction with track data is done through a `TrackView` object that is requested by a call to the `getTrackView` function of the `Track` object. A `TrackView` functions as a view over a given region of a `Track` and for that reason `getTrackView` takes in a `GenomeRegion` as argument. A `GenomeRegion` object holds information about genome and seqid, as well as regional information in form of a start and end base-pair position represented by two integers.

The `TrackView` class provides two ways of accessing data, either through iteration or by reading column slices into memory as NumPy arrays. In the former, the `TrackView` class yields `TrackElements` in the old GTrackCore and `PytablesTrackElements` in the new GTrackCore. `PytablesTrackElements` are linked with binary data through row objects of the track table. The iterator that yields the elements, sets their row attribute as soon as the row has been retrieved from `table.iterrows()`.

This entire composition can be seen in Figure 4.1.

4.2.2 Track preprocessor

The track preprocessor has played an important role in the mission to replace the existing ad-hoc data model with one that is based on use of a standardised package, being the component of GTrackCore where textual genomic data is exported to a binary data format.

Although the most significant code contributions can be found in the `OutputManager` – the module of the preprocessor in which the logic that concerns storage of data can be found – it is still important to give an overview of preprocessor altogether. Both because its structure has had an impact on how the `OutputManager` have been assembled, and because there has been done various changes and minor adjustments here and there.

A natural place to start is to explain is how textual tracks are parsed and managed in GTrackCore. We will then move on to describe how a preprocessing job is run, and present the steps that it involves. Afterwards, in Section 4.4, we will come back to the `OutputManager`.

4.2.2.1 Data directory structure

All the files produced or used by the preprocessor of GTrackCore are collected in a common data directory named `gtrackcore_data`, which in the new GTrackCore is located in a path decided by the `GTRACKCORE_DIR` environment variable. Figure 4.2 shows the directory structure of the old GTrackCore, and Figure 4.3 shows the directory structure of the new GTrackCore.

Metadata Metadata is found in the `Metadata` directory of the base path. The directory contains a single common *shelf* where metadata of all earlier

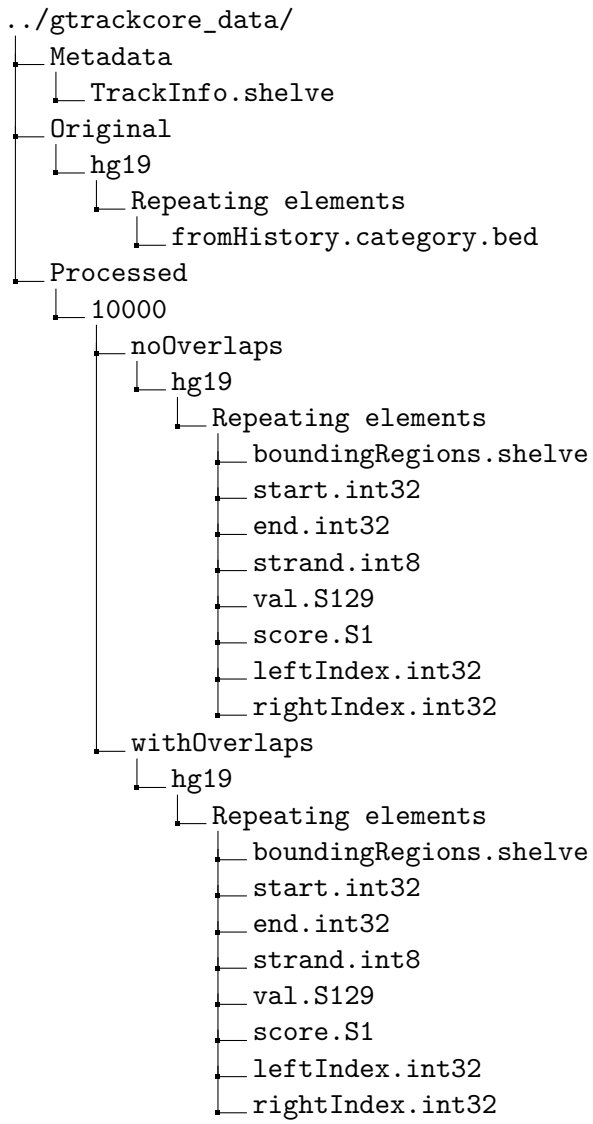


Figure 4.2: The directory structure of the old GTrackCore

preprocessed tracks are located. The key used to do lookups in this shelf is a combination of track name and genome name.

Original data Textual versions of genomic tracks are in GTrackCore collectively designated as *original data*, and are located in the `Original` directory of the base data path. Within it, the textual tracks are located in a path that is made up of a genome and a track name. The actual text files are located at the leaf level of the path. All text files located in the same leaf node are parts of the same track, as tracks are often divided per chromosome. When the preprocessor is called, it recursively scans this directory for a path that corresponds to the track name and genome given as parameters.

```
../gtrackcore_data/
├── Metadata
│   └── TrackInfo.shelve
├── Original
│   └── hg19
│       └── Repeating elements
│           └── fromHistory.category.bed
└── Processed
    └── hg19
        └── Repeating elements
            └── repeating_elements.h5
```

Figure 4.3: The directory structure of the new GTrackCore. The track contained in it is the same as in Figure 4.2.

Preprocessed data Is located in the `Processed` directory located in the base of the data path. The internal structure of this directory is the same as the `Original` directory, except that the files at the leaves are preprocessed tracks. The type of files that is located in this directory differs between the old and new GTrackCore.

Preprocessed tracks are in the old GTrackCore split into multiple files: A memmap for each of the columns of the track, a `leftIndex` and `rightIndex` memmap, and a shelf containing all the bounding regions of the track. In addition a number that states the bin size, which is used in indexing of the track, prepends the paths. All of these files are deprecated in the in the new GTrackCore.

In the new GTrackCore the `Processed` directory is less populated, and only contains a single PyTables file at leaf level. The informational content of this file alone is basically the same as all the files of of a single track in the old version, except for the `leftIndex` and `rightIndex` memmaps, which have been removed since they only functioned as internal index structures, and were used to locate bins.

4.2.2.2 How tracks are parsed

A central part of the preprocessing is to parse textual genomic tracks, or original track sources as they are referred to as in GTrackCore. Because of their potentially large size, the track sources are not read in their entirety in one run. Instead they are parsed element by element and fed to the rest of the system as they are requested.

In the next couple of paragraphs we will present classes that are important to the process of creating objects out of the genomic elements of the textual track sources, so that they later can be stored as binary data.

GenomeElementSource This abstract class, abbreviated as `GESource`, functions as an interface to a stream of genomic elements from a track, represented by the class `GenomeElements`. Since the class is abstract and provides iteration over generic genome elements, the actual parsing of the

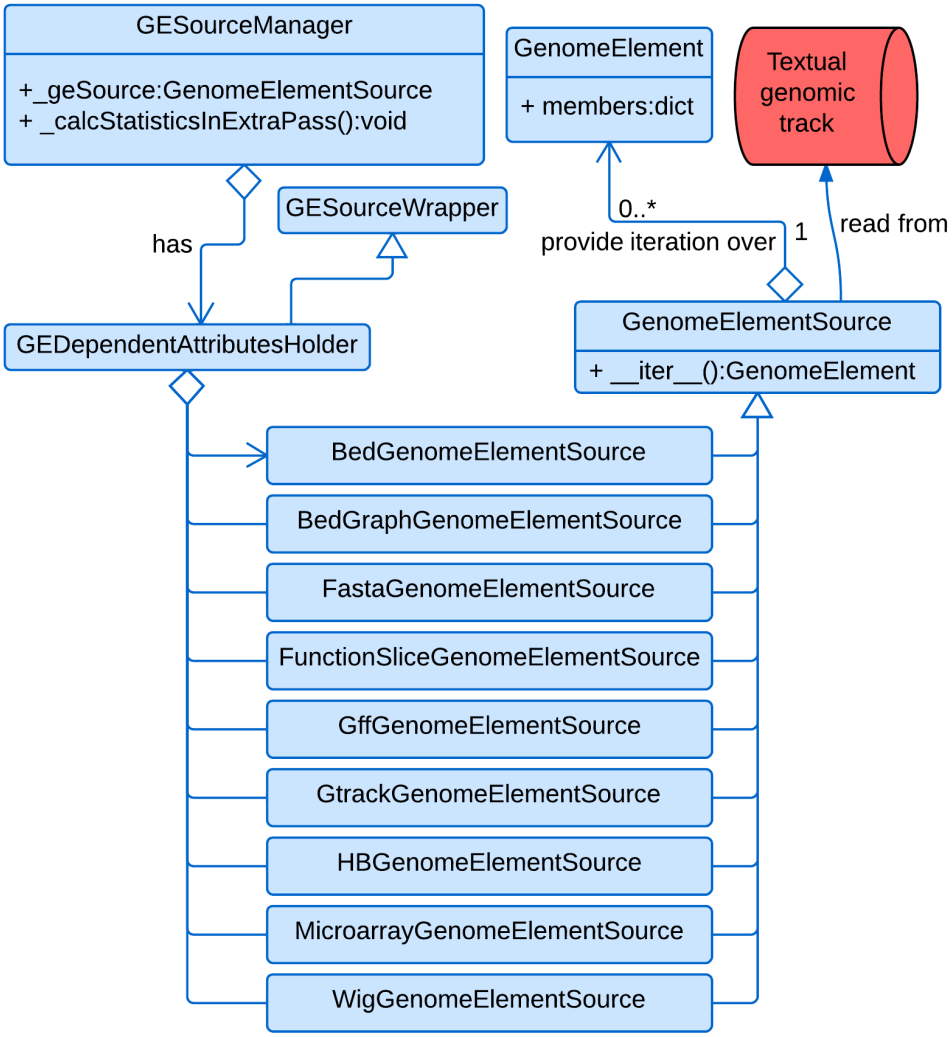


Figure 4.4: Diagram of classes involved in the extraction of genome elements from textual genomic data.

textual data is decoupled from the rest of the preprocessing code. The `GESource` also provides meta information about the source, such as the genome, track name, whether the source is sorted, etc. Classes that extend it must override these.

Distinct file types, such as BED and WIG, differ in formatting, and must therefore be parsed accordingly. This has been solved in GTrackCore by letting the `GenomeElementSource` class serve as a superclass to a variety of classes that are able to parse one textual file format each, but still output the same generic `GenomeElements`.

A schematic overview of the classes involved in parsing of textual track sources, i.e. extraction of genomic elements, is found in Figure 4.4.

GESourceManager This class encapsulates and manages a `GenomeElementSource`. In the same way as the `GESource` provides meta informa-

tion about the source itself, the `GESourceManager` provides metadata and statistics about all of the `GenomeElements` of the source. The statistics are calculated in the `_calcStatisticsInExtraPass()` method, by traversing all of the `GenomeElements` of its `GESource`, unless they are already cached.

Another important function of the `GESourceManager`, related to metadata, is the computation of bounding regions. Tuples of bounding regions are either extracted from the `GESource`, if they are explicitly defined there, or created on the go in the `getBoundingRegionTuples` method. The last option basically involves creating one bounding region for each chromosome that is defined for the track of the `GESourceManager`; the start bp locations are all 0's and the ends are found by doing a lookup in the `GENOMES` dictionary of the `GenomeInfo` class. The preprocessor later uses `BoundingRegionTuples` when the `BoundingRegionHandler` is created, and is one of the parts that have been rewritten in the new GTrackCore. We will come back to this in Section 4.3.2.

The class is not abstract and may be instantiated itself, despite having two subclasses. The most interesting one of these subclasses, the `OverlapClusteringGESourceManager`, will be addressed later.

4.2.3 Structure relevant to the new GTrackCore

4.2.3.1 Ordering of data within a PyTables file

An example of the internal object tree of a PyTables file containing track data can be found in Figure 4.5.

The directory structure where tracks were stored in the old GTrackCore (see Figure 4.2) served as a template for the internal node structure of PyTables track data files. The main reason for virtually using the same structure was to make the new GTrackCore somewhat backwards compatible with the old. The structure is based on how track names are formatted, i.e. each constituent of the name acts as category that may be shared among several different tracks. This in particular makes the structure highly suitable to use internally in PyTables files, for the same reason as in the file system: Since it allows having multiple tracks laid out in a single file by letting the them branch from the same root.

We have, however, branched the with- and no-overlaps trees at the uppermost group level, instead of at the root – as the tracks were ordered in the directory structure of the old GTrackCore. The motivation for this was to have the two overlap tables, which will be explained in Section 4.3.1, stored 'closer' together to express their affinity to each other, and to have a sensible place to store the persistent `TrackInfo` (see Section 4.3.3). Both were done to prepare for the expansion of storing multiple preprocessed tracks within the same PyTables file. A tool that demonstrates how this can be done will be presented in Section 4.6.2.

4.2.3.2 The Database module

To not pollute the entire GTrackCore package with code related to managing the internal node structure of PyTables files, an additional level

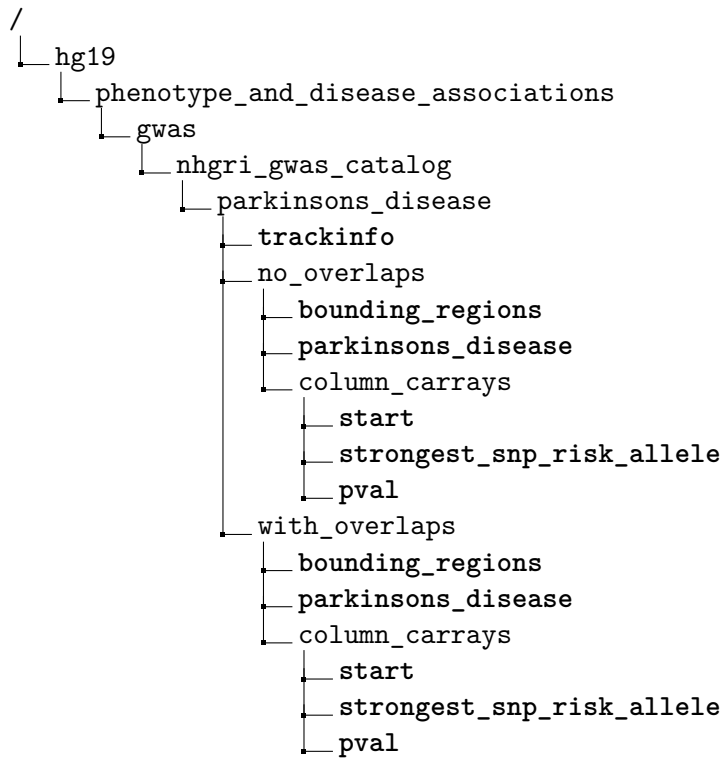


Figure 4.5: Example of ordering of data within a PyTables file, for a track named 'Phenotype and disease associations:GWAS:NHGRI GWAS Catalog:Parkinson's disease'. The track name is normalised to use PyTables 'natural naming'. The CArrays are added within the 'column_carrays' Group (see Section 4.3.2 for how and why these are created). All nodes marked in bold, except from the 'TrackInfo' metadata file, are Leaf nodes. The rest of the nodes are Groups.

of abstraction has been introduced. A class named Database handles the creation of Groups, acquiring of existing Nodes, creation and removal of Tables, etc. The Database class has two instantiable subclasses, the DatabaseReader and the DatabaseWriter, which respectively opens the PyTables file in read (r) mode after acquiring a shared lock, or in append (a) mode after acquiring an exclusive lock (see Section 2.6.1.4 for information about file locking in Python). They are both initialised by supplying the filename of the database that one wishes to open. Beyond their modes and lock types, Database subclasses differ in the variety of methods that are provided externally. It is not possible to make changes to a file in read mode, so all methods that are somehow editing the node structure, such as the remove_table method, are only found in the DatabaseWriter. The writer is thus, perhaps not surprisingly, the class that is being used when genome elements are being added as rows to a table in the OutputManager. The fact that the database writer primarily is used by the OutputManager is the reason why it is opened in append mode, and not simply in write mode. Write (w) mode overwrites existing data if there are any, and cannot be used

when the same database is opened many times and the desired functionality is to append elements, as we will see in Section 4.4.1.

This module went through a major refactoring during the project period. A description of this refactoring and its significance can be found in Appendix C.

Adding compression The `Database` module is naturally also responsible for compressing the PyTables file that it manages. Compression is activated by creating a filter that specifies the compression level and which compression library to use. The filter is then used as input to the open method, so that data passes through the filter on its way to and from the disk (see Section 2.6.3.3).

The option to add compression has been included to see how much effect it has on retrieval speeds and file sizes, the results of which will be seen in Chapter 5.

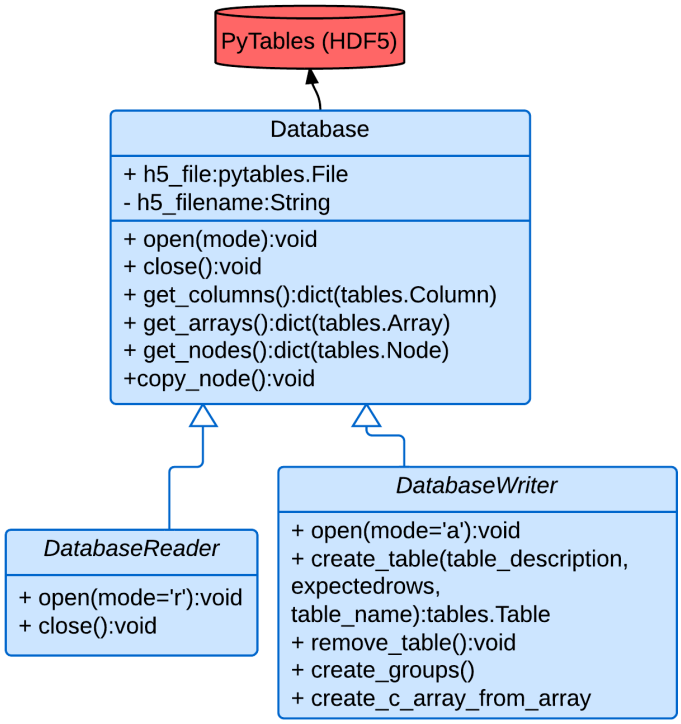


Figure 4.6: The Database class and its two subclasses.

4.3 Running a preprocessing job

The `process` method that initiates the compound preprocessing job is located in the `PreProcessTracksJob` class. Since the class is abstract, it is instantiated through one of its subclasses, which mainly differ in how they override the method that decides the type of `GESource` to use. The most basic and frequently used of these subclasses is `PreProcessAllTracksJob`, and preprocessing through this entity yields one `GESource` for each file that is associated with the track.

When the process method is called it enters a loop where each `GESource` is encapsulated in an object of the `GESourceManager` class, which wraps the `GESource` based on the track type, i.e. whether the track is *dense* or *sparse*, and the *phase* that the method is in.

4.3.1 Two preprecessing phases

The main loop of the `process` method can be viewed as a procedure that is divided into two phases:

1. **With-overlaps:** Genomic elements are read from one or multiple text files, and then stored as binary data.[1]
2. **No-overlaps:** Involves one of the following:
   (a) If the track is *sparse*, genome elements are read from the binary data created in phase 1 (see Figure 4.7). All possible overlapping elements are then merged and stored in a second binary data container.
   (b) If the track is *dense*, genome elements read in from one or more text files and stored as binary data, in the exact same way as in phase 1.

Note that the file product of the two phases are separated, and must be merged in the new GTrackCore. The solution to this is explained in Section 4.3.3.

The binary data is created in the `OutputManager`. We will, in Section 4.4, come back to how this class works in the new GTrackCore, i.e. how it is constructed and how elements are written through it. Prior to this we will look at the two preprocessing phases in a higher level of detail, and then detail the rest of the `process` procedure.

4.3.1.1 The with-overlaps phase

In this phase, elements are extracted from the original data of the track that is requested to be preprocessed, and then stored in the same way as the elements are laid out in the text files. As the name implies, possibly with overlaps, i.e. elements that have overlapping base-pairs positions (see Section 2.5.1.3). As noted in the phase listing in the previous section, this

[1] This phase is only executed for tracks that are *sparse*.

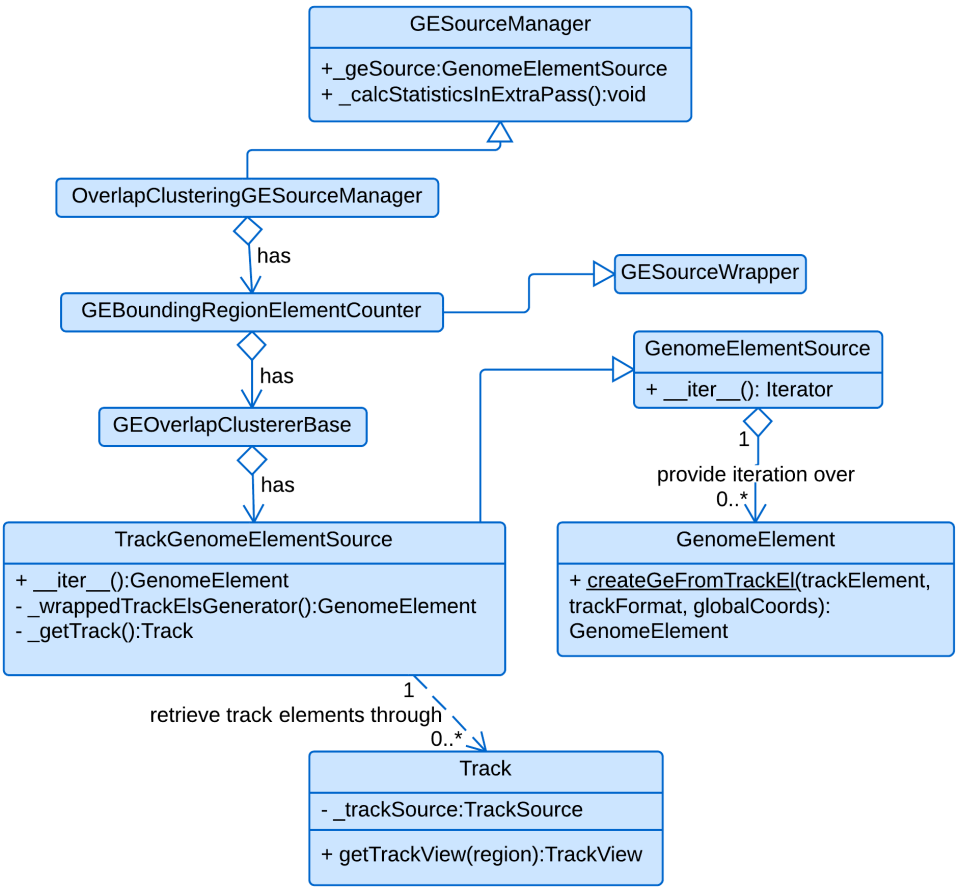


Figure 4.7: Diagram of classes involved in the extraction of genome elements from preprocessed data, without overlaps

phase is executed only when the track type is sparse, since dense tracks by definition do not have overlaps.

In the new GTrackCore the result of this phase is a temporary PyTables file named `<trackname>_with_overlaps`, located in the data path described in Section 4.2.2.1. This file contains a binary track table, a bounding region table, and a pickled Python object that holds the metadata.

In the old GTrackCore on the other hand, the result is several memmaps, a bounding region shelf and at most two index structures (see Figure 4.2).

4.3.1.2 The no-overlaps phase

Regardless of whether the track type is dense or sparse, the resulting binary track created in the no-overlaps phase is, in the new GTrackCore, a temporary PyTables file named `<trackname>_no_overlaps`, located in the `Processed` directory in the `gtrackcore_data` path.

In the old GTrackCore, the result is similar to that of the with-overlaps phase, only here contained in the `noOverlaps` directory without overlaps.

How the extraction of genome elements takes place for different track types is further described in the following paragraphs:

Sparse tracks For sparse tracks, the process of creating a track without overlaps involves extraction of genome elements from the preprocessed data of the with-overlaps phase. For this a special subclass of the `GESource` base class named `TrackGenomeElementSource` must be used. As we saw in Section 4.2.1, retrieval of data from preprocessed tracks must happen through a `TrackView` object, so the `TrackGenomeElementSource` creates a `TrackView` for each of the track's bounding regions. The `GenomeElements` yielded from the `TrackGenomeElementSource` are thus converted `TrackElements`, which are fetched by traversing all of the created `TrackViews`.

The overlaps are removed in a subclass of `GESourceManager` which is named `GEOverlapClustererBase`. As the name implies, overlapping elements of the source are merged into one, before its iterator yields them. Because of its complexity and the fact that we have not changed its inner working, this class will not be described in detail. We will rather look at it as a black box that does what it is supposed to. This is however where the majority of the work in the no-overlaps phase happens.

An important thing to note is that the file that contains the with-overlaps data must be kept open while the no-overlaps binary file is created. This is because the `OutputManager` is fed `GenomeElements` from a track source, meaning that elements are read and then stored, one after another. This design choice, done by the original developers of GTrackCore, have played a role in some of the decisions that were taken regarding how the PyTables binary data file product of the preprocessor is created. This is elaborated in Section 4.3.3.

Dense tracks In this case genome elements are extracted from the raw track data directly, in the same way as described in Section 4.2.2.2 and depicted in Figure 4.4. Dense tracks skip the with-overlaps phase, and have thus not extracted anything from the files that are requested to be processed yet. This is the reason why the no-overlaps phase for dense tracks involves the same as the with-overlaps phase for sparse tracks.

4.3.2 Local finalisation

Immediately after the end of both the no-overlaps and with-overlaps phases, after their respective track sources have been stored as binary data, a local finalisation step is executed. The first event, i. e. storage of bounding regions, occurs in both the new and the old GTrackCore. The second and third, i. e. sorting of track tables and creation of additional `CArrays`, only occurs in the new.

4.3.2.1 Storage of bounding regions

The bounding region tuples, which are created by the `GESourceManager`, has to be persisted so that they can outlive the preprocessing process and be used in track the data retrieval process. This takes place in the `BoundingRegionShelve` in the old GTrackCore and in the `BoundingRegionHandler`

of the new GTrackCore, respectively, using a ordered shelf and a track table as persistent data structure.

The `BoundingRegionHandler` is basically a direct port of the `BoundingRegionShelve`. This new replacement has to behave exactly like the old, also when something is wrong and the creation of the data structure should be stopped. The class performs sanity checks on the input bounding regions and throws exceptions if something is wrong. A few of the test cases located in the `test` modules asserts that these are raised, so it was important to make it work correctly in these situations as well. Most of the information is the same in both the new and the old data structure, except from *bin sizes*, which are left out from the new structure as they are not needed.

We originally put indices on the bounding region tables, but they were dropped due to a decrease in retriveal speed, rather than the improvement we expected.

4.3.2.2 Sorting of track tables

The `sort_preprocessed_table()` method is called as part of the local finalisation step in the new GTrackCore. The work that is done in this method is directly dependent upon the workings of the `OutputManager`, i. e. the module that creates the track tables. We will come back to this in Section 4.5, after we in Section 4.4 have detailed how the `OutputManager` of the new GTrackCore works.

4.3.2.3 Optional creation of CArrays

As an attempt to improve retrieval performance, in the case where portions of columns are *sliced* and read in as NumPy arrays, there has been added functionality that creates PyTables `CArrays` (see Section 2.6.4.2) directly from the created tables, i.e. multiple homogeneous arrays a single table of heterogeneous type. This is done by passing the columns of the table, created in either the with- or no-overlap phase, as input to the `create_carray` method of PyTables. The reason why we chose to create the arrays as part of the local finalisation step, instead of creating them alongside the tables was because we did not want to make a mess in the `OutputManager`. Assigning table columns to *CArrays* and adding them to the PyTables file, after the actual preprocessing, proved to be a fairly efficient operation despite having to copy entire columns.

Since arrays were added only to see whether the use of them would increase retrieval speed, it is optional to create them in the performance tools (see Section 3.4.3). Chunked arrays, i.e. `CArrays`, were chosen because we also wanted to test the impact of compression on retrieval times.[2] The results from a test of how long time it takes to preprocess with this 'array option' enabled, along with other performance results, are presented in Chapter 5. The results from the retrieval of arrays can be found in the thesis by Skifjeld [38].

[2]Only chunked arrays can be compressed, since the Blosc compressor is a HDF5 Filter (see Section 2.6.3.3).

4.3.3 Global finalisation

In the old GTrackCore, this 'global' finalisation step, performed after the with-overlaps and no-overlaps phases, primarily involved assigning metadata variables to the `TrackInfo` object, and to then store this in a metadata shelf. This is also done in the new GTrackCore, but here the process is more complicated due to the requirement to have the metadata reside in the same PyTables file as the track tables, as we will come back to below.

In the new GTrackCore, this finalisation step additionally includes *combining* the two track tables created in the with-overlaps and no-overlaps phases, which were temporarily put in separate PyTables files. This is done to exploit the capability of the HDF5 format to store multiple data sets in a single file.

4.3.3.1 Metadata inside the PyTables file

As we saw in Section 2.6.4.3, the `FileNode` module of PyTables makes it possible to put regular files into the object tree of a PyTables file. This feature is used to expand the database with metadata, by having a `TrackInfo` object reside as a pickled file at the second highest level of the group tree in the file. See Section 4.2.3.1 for how we order nodes within a PyTables file.

Problems with having metadata in the object tree Having the `TrackInfo` object reside in the same PyTables file as the track tables would not have caused problems if metadata writing was done exclusively by the track preprocessor. We were, however, informed by one of the original developers of the Genomic Hyperbrowser/GTrackCore that metadata may be written by any kind of process, i.e. by readers and writers alike. Retrieval processes are accessing PyTables files through a `DatabaseReader` instance (see Section 4.2.3.2 for how PyTables files are handled), preventing all other processes from writing anything.[3] The reason for this is twofold:

1. `DatabaseReaders` acquire a shared lock that prevents other processes from writing.
2. It is not possible to have a single HDF5 file opened concurrently by two different processes when one of them is in read mode and the other in write mode.[4]

Because of this, storage of a `TrackInfo` object in the object tree of a PyTables file requires a somewhat sophisticated solution, that is perhaps

[3] There might be multiple readers working on the same binary data files. For instance, if GTrackCore is coupled with the Genomic HyperBrowser in the future, several analysation jobs may be run concurrently.

[4] From the HDF5 FAQ:
'*Does HDF5 support concurrent access to a single data set from multiple processes?* If all processes are reading, then, yes, HDF5 (serial) does support this. If there are any processes that are writing, then no, this is not supported. We are working on a "Single Write Multiple Read" (SWMR) feature, which will be available in a future release (expected to be in HDF5-1.10)' [18].

more complex than desirable. The selected solution will be presented shortly, but first we will look at a more straightforward method that was considered, but proved to be a bad idea.

Wait util other processes are finished before writing A method that probably would not have required as much code as the one eventually selected, since only the `store` method of the `TrackInfo` object would have been altered, is summarised in the following steps:

- Close the current `DatabaseReader` temporarily.
- Open a `DatabaseWriter`, and wait until any other read processes are finished (and release their respective shared locks).
- Write updated metadata to the `TrackInfo` residing in the PyTables file.
- Reopen the `DatabaseReader` and continue to read.

This is not a satisfactory solution. Imagine you for instance have a large analysation job that takes days to finish, and a small one, estimated to finish in few seconds. If the small one is run before the large has finished, and the small one has to write metadata midways, it would have to wait until the large job has stopped. Readers may, in other words, have to wait indefinitely for other readers to finish.

A few other solutions similar to this method were assessed, but it was concluded that the one we eventually came to implement, which will be presented shortly, was the best we could come up with. That is, without having to do major changes to the existing code base. It was unfortunately not enough time for this, considering that work on the retrieval module had to be prioritised.

Using metadata shelves as 'dynamic' structures The 'old' `TrackInfo` metadata shelf, located in the Metadata directory (see Section 4.2.2.1), is used as a *dynamic* structure. Dynamic in the sense that it is used for reading and storage of metadata for active retrieval processes that are currently executing. Access to the `TrackInfo.shelve` object is restricted by the third-party module `safeshelve.py`, which basically locks the file (see Section 2.6.1.4), meaning that it is safe for multiple concurrent processes to write metadata to it.

The persistent `TrackInfo` metadata object, residing in the PyTables file, is first requested to be updated after the reader has finished. This is done by registering a method that updates the persistent metadata to the `atexit` module, meaning that it automatically is being stored upon termination of the interpreter (see Section 2.6.1.5). Shortly summarised, the method checks whether the dynamic `TrackInfo` object is dirty by comparing its `timeOfLastUpdate` variable to the one of the *static* `TrackInfo` stored as a `FileNode` in the PyTables file. Code related to this is located in the `MetadataHandler` module.

4.3.3.2 Combine the with-overlap and no-overlap files

Initially, it was not known that the with- and no-overlap tables should reside in the same file, and in early versions of the new GTrackCore the two tables were never merged. The with and no-overlaps branching was at this time done in the file system, in the same way as with the old GTrackCore (see Section 4.2). When we were made aware of the requirement, the desired solution was one that did not require much restructuring, and thus could be implemented quickly.

As explained earlier, the second phase of the preprocessor sometimes relies on extracting genome elements from the table created in the first phase. This complicates the goal of having the two distinct tables, i.e. . the with-overlaps and the no-overlaps tracks, and their associated bounding regions, stored in the same PyTables file. Hence, a more verbose and straightforward solution was chosen, where the with-overlaps and no-overlaps tables are separated up until the finalisation phase is reached. Here the `merge_and_rename_overlap_tables` method is called. This method essentially does what its name implies, and combines the content of the two files created in the previous phases. Copying nodes from one group to another is a fairly simple task in PyTables, and is done by invoking the `copy_node` method of a table. Because the method supports copying nodes between files, it is perfectly suited for the work of combining two files in their entirety.

Suggestion of having the tables in the same file during preprocessing A way of having the two tables in the same PyTables file from the start, could have been to use the same `DatabaseWriter`, i.e the same file node, throughout a preprocessing job. This is, however, not as simple as it may seem. What makes it complicated is that the preprocessor sometimes, when the original track source is sparse, relies on having binary data extracted from the retriever module. This means that the retriever module, when used by the preprocessor, would have had to use a common `DatabaseWriter` instead of a `DatabaseReader`, as it normally would. A way to accomplish this would be to override the `__new__` method of the `DatabaseReader`, and thus control its instance creation. If a state variable, '`is_preprocessing`', is set to `True`, a cached `DatabaseWriter` instance, dedicated to the preprocessor, is returned. Otherwise, a `DatabaseReader` instance is returned as normal.

Controlling instance creation such as this is referred to as the *singleton* design pattern. Use of singletons is often considered a bad practice, since they for instance break the renowned *single responsibly principle* [25] of object-oriented programming. The principle states that if a class has multiple responsibilities there will be more than one reason for it to change, which in turn will make it prone to problems if one of the responsibilities are changed. The changes may impair the class' ability to meet the other responsibilities, and may break the code in unexpected ways.

It is conceivable that the decreased number of opened `Database` objects that this solution would have resulted in, in turn could have made the preprocessor perform better. However, since the number of readers opened

in a preprocessing job is static and not decided by the size of the track, the decrease would have had a mentionable impact only on smaller tracks. Profiles of both 'Parkinson's' and 'Sequence' (consisting of 67 and 5232241 genomic elements, respectively), shows that the number of calls to the `open_file` method of PyTables (that is called each time a database instance is opened) is 333. This cumulates to 1.33 seconds. For 'Parkinson's', this amounts to approximately 25% of the overall running time of 5.40 seconds. For 'Repeating elements', on the other hand, it amounts to approximately 0.05 percent of the running time of 2493.90 seconds.

An argument for merging The suggested solution is, as the explanation of it probably has indicated, somewhat complicated and would have meant very little with regard to an increase in performance. Namely, approximately 1.33 seconds for any track since the number times a PyTables file is opened during a preprocessing is static. Considering that preprocessing normally is done once for each textual track, a minor speed improvement such as this is probably not reason good enough to initiate a complex refactoring, like the suggested solution would have required. It is also important to not underestimate the negative effects complex code can have on maintainability. The suggestion, which requires multiple changes throughout the code base, would likely have made the code base more 'tangled', making it more difficult to debug and make changes to later. The merge operation, on the other hand, is only an extra operation added to the local finalisation step, and does not have an impact on the structure of the rest of the code base.

Additionally, the cumulative time spent in the `merge_and_rename_overlap_tables` method has proved to have no significant effect on the running times of a preprocessing job, primarily because the method is only called once. A profile run of the sparse track 'Genes' showed that the merge operation in this instance had an impact of 1.686 seconds. This is approximately 0.7% of the total preprocessor running time of 247.35 seconds. Profiles of the two other sparse selected test tracks show consistent results. For these, the time spent on merging amounts to approximately 0.02% and 0.8% of their respective total running time. The merge method has, in other words, very little impact on running times.

4.3.4 Storage of genome elements

While the `process` procedure is doing the preliminary work of extracting genome elements from some track source, the elements are not requested to be written to a secondary storage device within its method body. This task is delegated to the `_createPreProcFiles` method of the `PreProcessGeSourceJob` class, which handles everything related to file creation. This method is called once for each `GESourceManager` of both the with-overlaps the no-overlaps phase, and does the same regardless of track type and overlaps, i.e. to traverse the iterator of the provided `GESourceManager` and then write yielded genome elements to a track. Information about whether there are overlapping elements is only used to determine *where* the elements should be written to, and does not influence

the steps that are undertaken to store them. The actual writing happens in an abstraction layer called the `OutputManager`, which from the outside generically handles output of genome elements. The interaction between all the aforementioned classes is depicted in Figure 4.8.

We will in the next section take a look at the inner mechanics of the `OutputManager`, and modules related to it. We start by giving a short overview of how the code related to the class is structured, before we move on to describe how it is set up. The section will be concluded with information about how genome elements are written to the set up table

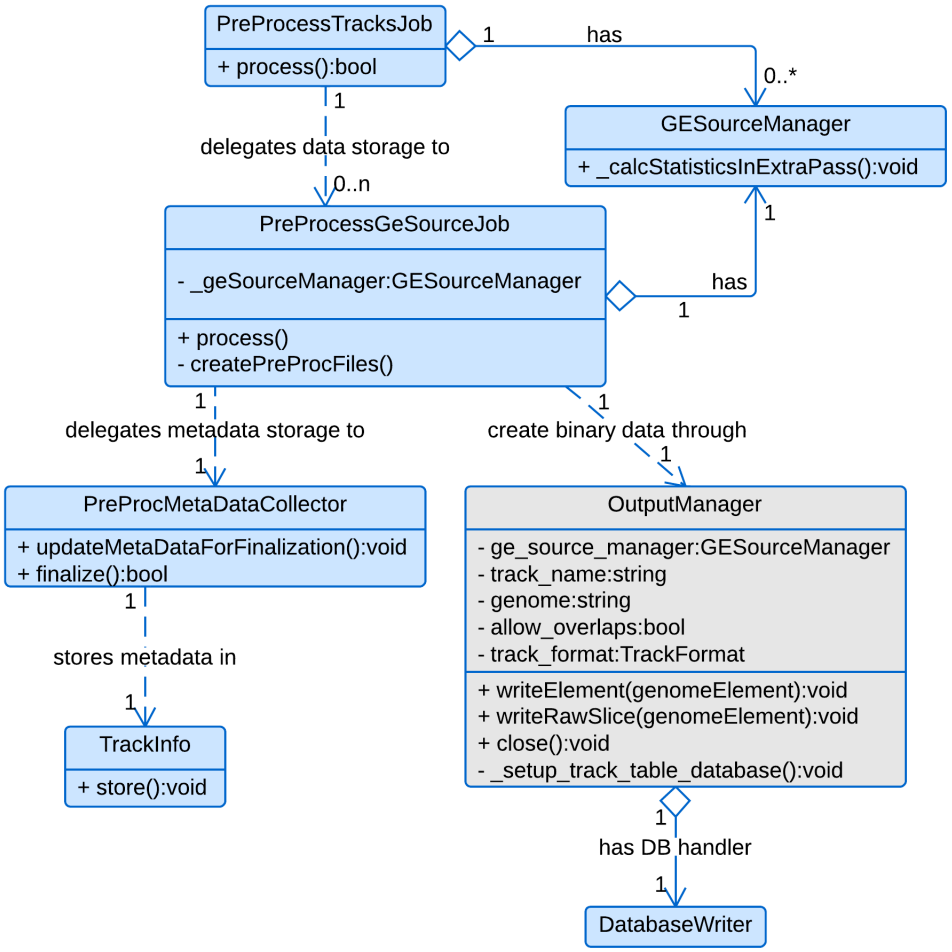


Figure 4.8: The interaction between the different modules that are related to storage of track data and metadata.

4.4 The OutputManager of the new GTrackCore

Storage of tracks is carried out in the `OutputManager`, serving as an interface for writing elements to disk. How its public methods '`writeElement`' and '`writeRawSlice`' are implemented have changed in the new GTrackCore, as they are not writing elements to memmaps anymore, but instead are writing

them as rows to PyTables tables.

The way the `OutputManager` is structured in the old GTrackCore reflects how the binary track data is ordered as Numpy memmaps, having one file for each column in a directory whose path uniquely identifies a track. Here the `OutputManager` contained an `OutputDirectory` object, which again contains an `OutputFile` object for each column of the track. This composition has been simplified in the new GTrackCore, and the directory class and the file classes have been left out, due to all the columns being stored in a single file.

All code related to this class bas been organised in a package named `pytables` to explicitly state that it contains logic related to storing tracks using PyTables. This package includes a helper module named `CommonTableFunctions` that contains the methods for manipulating existing tables, e.g. methods for *sorting* and for *resizing* of track tables. It also has a module named `TableDescriber` that collects code concerning PyTables *table descriptions*. We will come back to the significance of these in Section 4.4.1.1

4.4.1 Setup of the OutputManager

The setup methods called upon initialisation of the `OutputManager` are preparing the object to write elements to a table. This task is rather complicated and must take a few special cases into account. The fact that a new instance of this class is created for each `GESourceManager` means that its setup either involves creating an entirely new table, or it involves resuming an ongoing table construction, by doing some alterations to the existing columns.

Before any of these two paths of setup can be commenced, a *table description* object must be made. This object defines the desired structure of the data that is to be inserted into the table. Thus, we will in the next section start by giving a description of what this data structure contains and how it is created. We will then describe the two different setup paths in detail. Since storage of tracks always start by *creating a new table*, we will begin with this process. We will then explain how an ongoing table construction is *resumed*, or more specifically, how we handle columns that in a subsequent source, e.g. another original track file, have had properties such as shape and datatype altered.

4.4.1.1 Creating a table description

Since the individual columns of a PyTables table are of homogeneous type, their data types and properties must be clarified before any genome elements are written to the track table. This in particular is the purpose of a table description, which, as its name signifies, describes the informational content of a table.

There are in fact two ways to create a table description in PyTables, either by making a subclass of `tables.IsDescription` or by using Python's built-in dictionary type. We went with the latter since the former has no any

apparent advantages. The column names are used as keys in the dictionary. The values associated with these keys contain information about data type, and other properties such as the column positions (`pos`) or the default cell content (`dflt`). To define a table with the columns 'chr', 'start', 'end' and 'val', one could for instance create the following description:

```
table_description = {'chr': tables.StringCol(5),
                     'start': tables.Int32Col(),
                     'end': tables.Int32Col(),
                     'val': tables.Int8Col(shape=(2,1), dflt=-1)
                    }
```

In this snippet we see that the different columns get their PyTables data type class decided based on their informational content. Chromosomes are typically N bytes strings, and base-pair positions are represented as 32-bit integers. These data type classes are not decided entirely statically, since it for instance would have been a waste of storage space to use 100 bytes to store every 'chr' column, only because this is the largest allowed value for any track. This is where the `GESourceManager` comes to use. Maximum string lengths and data types can be acquired from the statistics contained by this class. Also *shapes* are implicitly defined there.

Note that the 'chr' column, used to represent bounding regions, is included only in tables that have 'start' or 'end columns. This is a temporary column that is used as an aid to when sorting track tables. This process will be detailed in section 4.5.

The functionality for deciding data types is also implemented in the old GTrackCore, and is used to define the structure of the column memmaps. The logic is, however, different and had to be adapted to work with PyTables in the new GTrackCore.

Shapes Are represented as tuples in PyTables, as they are in NumPy. The columns that may have a shape are only the 'val', 'weights' and 'edges' columns. All the other columns are scalars and have their internal shape set to an empty tuple. Note that column shapes are set as a requirement to how single cells should be formatted, not whole arrays as is the case for regular NumPy `ndarrays`. Therefore, the vertical dimension that corresponds to the total number of rows altogether is not included when the column shapes are defined.

Value The shape of the value column is simply based on the value dimension variable:

```
val_shape = val_dim if val_dim > 1 else ()
```

As we see, if the val_dim acquired from the statistics is either 0 or 1, the shape is set to an empty tuple. Otherwise, the value is a list and its shape is set according to the dimension.

Edges The shape of the edges column is calculated by using the maximum number of edges:

```
edges_shape = max_num_edges if max_num_edges > 1 else (1,)
```

Edges are always represented as a list, even when a scalar. Thus, if the number of edges is 0 or 1, a single dimension is created, which respectively holds nothing or a scalar.

Weights The shape of the weights column is calculated by using the maximum number of edges and the data type dim:

```
weights_shape = tuple([max(1, max_num_edges)] + ([data_type_dim] if
    data_type_dim > 1 else []))
```

The weights column is represented as a two-dimensional array. The first dimension represents the edges, and the second dimension the weights that are associated with each edge. Each edge may in other words have weights that are vectors.

4.4.1.2 Table creation

A new Table is created by a single call to the `create_table` method of the `DatabaseWriter`, which has the following list of parameters in its signature:

- A list of nodes specifying the desired `Group` of the new table. The list must be an amalgamation of the genome, the track name and a boolean variable specifying whether in the with- or no-overlaps phase.
- The expected number of rows, which is internally used by PyTables to decide the shape of the chunk that are to be written in a single HDF5 I/O operation. We looked at this in Section 2.6.4.4. The number used is the total number of elements of the current `GESourceManager`.
- A *table description* that gives a semantic description of the columns. The description is, as we saw in the previous section, a dictionary and is created in the `TableDescriber` module.

Inside its method body, the groups that do not exist yet are recursively created. Afterwards the equally named `create_table` method of the h5file object is called. To specify the location of the new table, the outermost group object is passed as argument along with the expected number of rows, the desired table name, and the description dictionary.

4.4.1.3 Resuming an ongoing table construction

As seen before, binary tracks are incrementally created. An `OutputManager` is created for each `GESourceManager`, and the genome elements of their respective sources are one after another stored in the same track table. For the first `OutputManager` this involves the creation of the actual table.

Immediately, it may seem like subsequent managers only have to acquire this table and then simply write their own elements to it. However, resuming an ongoing table construction is not always that trivial. The table description that defines all the columns of the table is exclusively based on statistics acquired from from `GESourceManager` of the `OutputManager` that created it. Upon the creation of the table, nothing is known about possible subsequent sources. This means that when the track table construction is resumed, the column descriptions may be outdated. Although the actual columns and their basic data types have to be the same across all the `GESourceManagers` of the same track, string lengths and array shapes may differ among different sources. A subsequent source could for instance have a 'seqid' column with longer maximum string length or a 'weights' column with an extra dimension. If this happens, the descriptions of the affected column must somehow be *edited*.

Optimally, PyTables would have had built-in support for editing column descriptions of an existing table, but unfortunately it does not.[5] If a column needs its itemsize (i.e. string length) or shape enlarged, a whole *new* table with an updated table description has to be created. Then all the rows of the old table has to be copied to it.

Update table description The updated table description, used to describe this new table, is created as a 'merged' result of the table description created for the current `OutputManager` and the description of the old table. We will refer to these as the *new table description* and the *old table description*. The merging is conducted in the following manner:

1. Find all the columns that might need to have a new shape or itemsize calculated. This includes 'id', 'val', 'edges', 'weights', and possible 'extra' columns (which always are strings and hence have itemsizes)

2. For each of the previously found columns, compare the column descriptions of the new table description to the ones of the old table description. If either or both of the shape and itemsize properties are larger in the new table description, a new column description with updated PyTables data type values is put in an initially empty `new_column_descriptions` dictionary

3. The new table description is finally updated with the `new_column_descriptions`, and is immediately used to create a new table.

The central part of this process is to decide whether an itemsize or shape is larger in the current `OutputManager`. Itemsizes are, as we have seen, integers that are used to specify the number of bytes that should be used for data types of non-fixed size, so to decide which one of these to use simply means to choose the higher of two integers. Getting updated

[5] This is a restriction of the underlying HDF5 file format, since datatype and dataspace (the dimensions) cannot be changed for the lifetime of the data set. See section 1.2.3 of [19].

Listing 4.1: Method for inserting an array into another which it does not fill completely

```
def insert_array_into_array_of_larger_shape(array, shape):
        new_array = numpy.empty(shape=shape, dtype=array.dtype)
        new_array.fill(get_default_numpy_value(array.dtype))
        new_array[[slice(0, shape_dimension) for shape_dimension in
            array.shape]] = array
```

shapes is a somewhat more complicated task. If the old column description of a 'weights' column for instance has the shape $(1, 4)$ and the new column description has the shape $(3, 2)$ the resulting shape should be $(3, 4)$. That is, the resulting shape should use the highest value for each dimension found in any of the two tuples. We have implemented this and all the other functionality for updating column descriptions in the `TableDescriber` class.

Create a new 'resized' table All the elements or rows of the old table must be copied into the new one. This is done in the `copy_content_from_old_to_new_table` method. The new table is created as we saw earlier, but instead of only passing along the number elements of the current `GESourceManager` as the `expectedrows` parameter, we add this number to the number of elements residing in the old table. The copying is done by looping through the rows of `old_table.iterrows()` and then add these to the new table.

An extra care must be taken when moving rows, which concerns multidimensional cells with an array shape that is smaller than the specified PyTables data type. If the assignment is done directly, such as when not dealing with shapes, something similar to the following will occur under the hood:

```
cell1 = np.zeros((2,3))
cell2 = np.zeros((3,3))
cell2[:] = cell1[:]
```

This will lead to a 'ValueError: could not broadcast input array from shape (2,3) into shape (3,3)'. The NumPy general broadcasting rules state that two dimensions are compatible if they are equal or one of them is 1, and in this example they are obviously not compatible since their first dimension differ. The proper way to do this kind of assignment, where the desired is to insert a smaller array into a larger one, is to 'transform' the smaller into a new array with correct shape before insertion. Our implementation can be found in Listing 4.1

4.4.1.4 Can the resumption process be made easier?

As has been shown, `GESourceManager` contains statistics only for the source that it manages, and knows nothing of previous or subsequent sources. This

means that shapes and itemsizes of subsequent sources are unknown when the table is created, and a resumption hence has involve to create an entirely new 'resized table, as we saw in the previous section. This solution, which may require several copy operations, was forced upon the new GTrackCore as there was no way to achieve the same in the general case. That is, to make all subsequent track sources fit in the same table regardless of shape and itemsize, without having to do large structural changes to the modules responsible for parsing of textual genomic tracks.

The resumption process would have been trivial if all the different track sources were taken into account when the table first was created, i.e. that shapes and itemsizes were calculated as a sum of all the forthcoming track sources. For this to be possible the parsing mechanism would have to be changed so that only a single `GESource` is used, also when a textual genomic track is divided into several files. The `GESource` would then have provided iteration over every genomic element contained in the track, not only over the elements of a single constituent. This would in turn have meant that the `GESourceManager` could have calculated statistics applicable to the entire track, so that a single `OutputManager` could have been used to store all the genomic elements of the source.

As doing changes to the parser of GTrackCore was out of the scope of our work, there has not been made an attempt at making the resumption process easier by doing the aforementioned suggested changes outside of the `OutputManager`. This is, however, something that can be done to increase overall performance of the preprocessor as the underlying copy operation can have a rather large impact on running times, especially if performed several times on a large track, e.g. a textual track consisting of multiple files, where each file has a shape or itemsize larger than the one that came before it. Although it is unlikely that this will occur very often, it can nonetheless happen, and the suggested changes to the parser could in these cases have resulted in a speed improvement. Then again, whether updating the parser is something that is worth spending valuable development time on depends on how important it is to quickly be able to preprocess this exact type of track, which again depends on how performance critical the preprocessor is defined to be. As has been mentioned before, the preprocessor is normally only run once and is thus not as performance critical as for instance the retrieval module. It is, however, important that it does its work within reasonable time.

The previously suggested changes, which probably will make the resumption process easier, are further explained in Chapter 7, where different areas of future work is presented.

We acknowledge that none of the 'selected test tracks' can be used to directly test how much time that is spend in the `copy_content_from_old_to_new_table` method for very large tracks. The track 'Interconnections' is the only track for which the method is invoked, but because of its relatively small size, i.e. 3 076 genomic elements, the profile of it does not give a good indication of how much of an impact the copying actually may have on running times. The underlying PyTables sub-procedure, however, where the actual copying is taking place, is tested for tracks of larger size through

the itersequence-based sorting method. This method will be discussed in the upcoming Section 4.5.1, and the profile results that will be presented there are applicable to a single table 'resizing'-process as well.

4.4.2 Writing genome elements

The genome elements are written as rows either through the `writeElement` method or the `writeRawSlice` method. Which of these that are used depends on whether the genome elements come from a 'slice source' or not. If the 'slice source'-boolean of the `GESource` is set, it means that the attributes of its genome elements are `ndarrays`. The main reason for using slices sources is to increase performance for large data sets. Otherwise, if the 'slice source'-boolean is `False`, the attributes are normal Python objects. Both methods take in and handle a single genome element at a time.

4.4.2.1 writeElement()

This is the most regularly used method to write elements, and is called for sources that have the 'slice source'-boolean set to false. The single thing it does is to call the `_add_ge_dict_as_row` method with a dictionary that contains all the attributes of the input genome element as parameter. The attributes are here inserted into a new row which is acquired from the track table. The row is then appended to the table by using `new_row.append()`.

For most genome elements the attribute is assigned directly to the new row, but for the three list attributes – 'val', 'edges' and 'weights' – it involves something more. For these, one first has to convert the list attribute to an `ndarray`, before its shape is compared to the shape of the column description. If the newly created `ndarray` is smaller than the shape of the column description, it has to be inserted into an array of a larger shape. Thus, we use the same method as described earlier (see Listing 4.1).

4.4.2.2 writeRawSlice()

Only when the genome element passed as argument has a single 'val' value attribute is this method doing something unique compared to `writeElement`. In that case the `_add_slice_element_as_chunk` is invoked with the element's 'val', which in the 'slice source' case is always an array. This array is written through `table.append()` which allows appending a block of rows to a table. Array slices of genome elements with attributes other than 'vals' are simply traversed and added through the regular `_add_ge_dict_as_row`. Hence not resulting in increased performance.

While no other class than the quite recently committed (6th May 2014) version of the `FastaGenomeElementSource`[6] has the 'slice source'-boolean set, and hence is using the `writeRawSlice`, we have still implemented it. There are, however, some performance issues related to the use of it, compared to the same method of the old GTrackCore, that will be addressed when the performance results are analysed in Chapter 6.

[6] https://github.com/brynjagr/gtrackcore/commit/2bd4a64e26afd6807e8e5297bcb79df02a66e8eb

4.5 Sorting track tables

Genome elements are most commonly residing in sorted order in the textual genomic tracks. However, it is not required that they are. The mechanisms for retrieving track data relies on having the genome elements sorted within their own bounding region. Hence, in order to ensure that elements always reside in the correct order in a track table, the tracks are sorted as part of the local finalisation step (see Section 4.3.2). Function, Linked Function and Linked Base Pairs tracks are guaranteed to be sorted, and are therefore not touched by the sorter, as a consequence of them not having any positional information, i.e. neither start nor end columns. Another situation where a track is not sorted is if the `GESource` is a GTrack file that has the 'Sorted elements' header variable set to `True`.

Note that we do not sort the bounding regions themselves, although input files may be read of in unsorted order. For example, a file named 'chr9.bed' is handled after 'chr10.bed' as a consequence of the fact that files are acquired from the `Original` directory in lexicographical order. This implies that elements from chromosome 10 are written to the track table before those of chromosome 9. This is not important, as the bounding regions are stored in the `BoundingRegionHandler` after the track that they belong to have been stored in the OutputManager. That is, the bounding region table will contain the correct track table indices even though if the regions are not residing in natural order.

PyTables does not have built-in functionality for sorting tables, so this had to be implemented. Throughout the duration of the project the track table sorting method has changed once. It was originally decided upon, and implemented, a solution where tables were sorted entirely in-memory, but because of a problem related to doing column assignments in PyTables we instead went with one that is based on iteration. This problem will be further elaborated in the upcoming Section 4.5.1, Common for both sorting methods, however, is that they first find the *sort order* in-memory by using `numpy.lexsort`. Underneath is an example of how the sort order is found for tracks that have 'start' and 'end' columns, in addition to the obligatory 'chr' column:

```
chr_column = table.cols.chr[:]
start_column = table.cols.start[:]
end_column = table.cols.end[:]
sort_order = numpy.lexsort((end_column, start_column, chr_column))
```

The reason why the 'chr' column is included in the sort order is to prevent elements of different chromosomes from being shuffled. After the order is found the 'chr' is not needed anymore as it resides in the bounding region table, and is thus deleted from the table. Note that the sort order that `numpy.lexsort` method gives is in backwards order compared to the order of the input tuple: The sorting is first performed on the 'chr' column, then on 'start' and finally on 'end'.

4.5.1 Assigning sorted ndarrays to table columns

The first sorting method that was implemented was based on assigning sorted `ndarrays` to table columns. The columns of the table were here, one by one, read into memory as `ndarrays` and then sorted in-memory through a slice assignment, by using the previously found sort order. The following one-liner is an example of how a 'start' column was sorted using this old method:

```
table.cols.start[:] = start_column[sort_order]
```

This method of sorting is similar to how tracks were sorted in the old GTrackCore, which instead of setting a PyTables column set the contents of a memmap. However, because of a decision made by the developers of PyTables it proved to be impossible to assign a `numpy.ndarray` to a complete column in the general case. The following, which is an excerpt from a sub-procedure of the `__setslice__` method located inside of the `Column` class of PyTables, put a stop to this:

```
# Get rid of single-dimensional dimensions
column = column.squeeze()
```

As the comment states, this line will 'get rid of single-dimensional dimensions', which affect columns with a shape. As we remember from the Shapes paragraph in Section 4.4.1, columns may have single dimensional dimensions intentionally, so it is not desirable to have these stripped away. Getting rid of single-dimensional dimensions will further along lead to an ValueError stating that a broadcast is not possible to carry out.

The issue has been reported on GitHub. The issue report, which is attached in Appendix D, includes an example of how slice assignments work in NumPy and how PyTables behaves differently in equivalent situations.

4.5.2 Utilising the itersequence method

Because of the problems with the aforementioned column assignment, we instead decided upon a sorting method that creates a new table and sequentially copies the rows of the old table to it. The sort order is used as input to the `itersequence` method of the old table, which gives an iteration over the rows in sorted order. The rows are then directly added as new rows to a new table like this:

```
new_row[col] = old_row[col]
```

How this sorting is performed, by copying rows from one table to another, is quite similar to the earlier mentioned 'resize' functionality. As a consequence, these methods have some of the same subroutines. However, as the new table in this case is using the exact same table description as the old, no extra care has to be taken when copying row fields with shapes.

4.5.3 Column assignments might be faster

To compare the two sorting methods, a profile of the new GTrackCore, where the old column assignment-based sorting is activated[7], has been run on the track 'Sequence' (the original track source is a single file, thus possible to do a column assignment). The profile shows that the cumulative time spent in the `sort_preprocessed_table` method amounts to approximately 20.022 seconds. This is somewhat better than the current default itersequence-based sorting: A profile of this sorting method[8], of the same track, shows that the cumulative time spent in `sort_preprocessed_table` amounts to approximately 172.420. This suggests that column assignment-based sorting might be as much as $172.420/20.0229 \approx 9$ times faster than itersequence-based sorting, although it probably depends on the size of the track. More testing should admittedly be done to confirm this result.

4.5.4 Estimated memory usage of the two methods

A variable that has not yet been taken into account, perhaps slightly in favour to the itersequence-based sorting, is memory usage. The column assignments happens entirely in-memory, whereas sorting by use of the itersequence method is partly disk-based.

Common for both sorting methods is that they depend on finding the correct sort order by use of `numpy.lexsort`. This process is memory-intensive, especially for large tracks, since at most three different columns have to reside in memory at once, i.e. the 'chr', 'start' and 'end' columns. Let us look at an example close to a worst case[9]: A Segments track that has genomic elements of length one defined for each base-pair of the human genome, i.e. 3 209 286 105 elements (GRCh38), where the three column amounts 13 bytes (two four byte integers and one 5 character string). Finding the sort order for this track would in theory have required $(3209286105 \times 13)/2^{30} \approx 40$ Gb of RAM, to be able to hold the columns as `ndarrays` in memory. On a dedicated server with very high specifications, such as the server where our Genomic HyperBrowser instance is running, 40 GB of RAM is low enough for it to not be a problem (the server has 500 GB of RAM). However, if GTrackCore was to be run on personal computers, as a command-line based tool, this amount of memory usage far exceeds what is currently possible. Tracks as large as in this example, which have 'start' or 'end' columns and thus have to be sorted, are, however, not very common and most tracks do not require that much memory. For instance, to find the sort order of a Segments track with 2.5×10^8 genomic elements, which is still a large track, would have needed $(2.5 \times 10^8 \times 13)/2^{30} \approx 3$ Gb of memory. Something that, unlike the previous example, would have been possible to do also on a lower-end computer.

[7] Assignment-based sorting activated: https://github.com/brynjagr/gtrackcore/commit/725da55a36a0f2c3e1a1165274da30814bf934cf

[8] Itersequence-based sorting activated: https://github.com/brynjagr/gtrackcore/commit/30ee9b6fc48555495d0c4741dbf07edd6a77bd6e

[9] There is actually no limit to how large a track can be, and thus no quantifiable worst-case. This is because different elements may be defined over the same coordinates

What separates the two sorting methods, in terms of memory usage, is what happens after the sort order has been found. The itersequence method is disk-based in the manner that the sort order is used as coordinate input to an iterator that yields elements in sorted order. The elements are, one-by-one, copied into the new table, meaning that nothing of significant size has to be held in memory. The column assignment-based sorting method, on the other hand, has to do the actual sorting in-memory as well, by reading each column sequentially. This can be a more expensive process than the preliminary work of finding the sort order. If, for instance, a track with 2.5×10^8 genomic elements had a column with a 100 byte string as data type, it would have taken $(2.5 \times 10^8 \times 100)/2^{30} \approx 23$ Gb of memory to sort it. This is about seven times as much memory as it takes to find the sort order. However, for tracks with fewer elements, in-memory sorting of a 100 bytes column would have been negligible in terms of memory usage. For example, a small/medium sized track with 10^6 genomic elements would have required $10^6 \times 100)/2^{30} \approx 0.01$ Gb of memory to be sorted.

4.5.5 Column assignment issue of PyTables worth fixing?

The most important difference between the two sorting methods is arguably running times. Memory usage is, as has been pointed out, not critical for any of the two methods when working on small to medium sized tracks. While the itersequence-based method is able to handle sorting of columns of larger data type than the column assignment-based method, both sorting methods would have encountered problems when performed on large enough tracks. This due to the expensive process of calculating the sort order.

As has been discussed earlier, the preprocessor of GTrackCore is not extremely performance-critical. Notwithstanding that high performance is desirable to some extent, since it means that textual tracks can be exported to binary data faster and thus be analysed sooner. Therefore, if PyTables and HDF5 prove to be a suitable replacement to the ad-hoc memmap-based data format, as will be discussed further in Chapter 6, it could be beneficial to investigate the possibility of having the previously mentioned column issue of PyTables fixed. One of the main developers of PyTables has for instance stated that a pull request that fixes the issue in the general case is welcome.

4.6 Additional tools provided

As a proof of concept, to illustrate some of the benefits of storing tracks as HDF5 data, we have provided two additional tools. One of them, called the GTrackSuite, lets the user acquire track data from local or remote sources, and then store it in the `Processed` directory. The other is an API for merging track tables, by utilising the ability to store multiple nodes in a single PyTables file.

4.6.1 GTrackSuite

The idea behind this tool was to allow for convenient exchange of genomic data, through a range of transfer protocols. Both to easily be able to acquire data from remote or local sources, and to actively share data with others.

The GTrackSuite could have been created for the old GTrackCore as well, but it would have been more complicated. All files that belong together would have had to be collected in an archive with a structure that resembles the local `gtrackcore_data` directory. In the new GTrackCore a binary track and all its associated data is located in a single file, which can be transferred directly without any further work.

4.6.2 mergeAPI

Having multiple tables in a single PyTables file is a neat way of organising tracks. It also opens for the possibility of expressing that there is a relationship between different tracks residing in the same file. The mergeAPI enables the user to both *merge* track tables located in different databases, and to *extract* a single track table from a PyTables that contains several. The tool additionally supports to extract several tracks of a single PyTables file, placing these in the correct subdirectories of `Processed` directory (see Section 4.2.2.1). What makes this possible is the fact that the internal group structure of the files has the same structure as the directory.

4.7 Summary

In this chapter we have presented an implementation of a PyTables-based preprocessor in the GTrackCore package. We have looked at how the preprocessor of the new GTrackCore produces two different track tables, namely the with- and no-overlaps tables, a bounding region table and a pickled `TrackInfo` object that holds metadata about the track. All stored together in a single PyTables file. Further we given a thorough description of the `OutputManager`, where the actual work of creating the track tables takes place. Here, we have described how new tables are *created*, how an ongoing table creation is *resumed*, and looked at two ways genome elements may be *written* to a table, i.e. either through `writeElement` or `writeRawSlice`. Subsequently, we looked at how *sorting* of tables is carried out, and at how the current sorting method has been chosen due to an issue with how PyTables handles columns. Lastly, we presented two proof-of-concept tools that demonstrate some of the advantages of using PyTables and HDF5.

4.7.1 Source code overview

An overview of the most important modules that have been created in the process of incorporating PyTables in GTrackCore can be found in Table 4.1. There are a few more files that have been created or changed, but the ones found in the table are those believed to be the most important.

We recall that the GitHub repository (see Section 3.5.2.1) that contains the new GTrackCore is made available in Appendix B.

4.7.2 Statement on the quality of the code product

There has been made an attempt at making the code robust, and self-explanatory by giving descriptive variable, method and class names, with the goal in mind of making it easy for future developers to understand the semantics of the code.

While verbosity has been prioritised over 'clever' solutions with quirky semantics, there are admittedly some examples of solutions that were not optimal, but arguably had to be done in the way they were due to design choices taken prior to the work on the new GTrackCore commenced. For example, the solution for having metadata stored in the object tree, that was presented in Section 4.3.3.1.

In regard to the test cases that were supplied, the new GTrackCore supports all 50 of them. In addition, the system is confirmed to support an extra requirement that the existing test cases did not cover, that we had to provide our own test case for. We will come back to this in Section 6.3.1. All the unit tests are passing as well, although a few of them have been deactivated since they were not relevant to PyTables.

4.7.2.1 Snake case naming convention

Our code contributions have been written using the snake case naming convention, i.e. compound variable and method names have their elements separated by a single underscore character ('_'), (e.g. 'somewhat_long_variable_name'). The rest of the code base uses camel case, i.e. all compound variable and method names have their elements start with a capital letter (e.g. 'somewhatLongVariableName'). While the recommend practice is to comply with the coding standards set by the project, we were told by the system administrators that we were free to refactor the code base to use snake case, which is the recommended naming convention of PEP8 [30]. However, because it was found beneficial to have our contributions distinguishable from the existing code, the rest package has not been refactored yet.

Table 4.1: The modules that have been edited for the implementation of the new GTrackCore. Files marked with an asterisk are directly relevant to the *preprocesor* and have either been created or completely rewritten.

Filename	Content
gtrackcore.preprocess	
PreProcessTracksJob.py	The main preprocessing loop, that was described in Section 4.3
PreProcessGeSourceJob.py	Extracts metadata from the `GESouceManager`, creates an `OutputManager` and uses it to write all the genome elements, as described in Section 4.3.4
gtrackcore.preprocess.pytables	
CommonTableFunctions.py*	Various method related to manipulating tables. E.g. the merger (Section 4.3.3.2), resizer (Section 4.4.1.3) and sorter (Section 4.5.2)
OutputManager.py*	Where binary track tables are created, as described in Section 4.4
TableDescriber.py*	Functionality for creating and updating table descriptions, as described in Section 4.4.1
gtrackcore.track.pytables	
BoundingRegionHandler.py*	Bounding regions functionality. Both creation of table and retrieval (see Section 4.3.2.1)
TrackSource.py	Creates `VirtualTrackColumns` and stores them in a dictionary
TrackViewLoader.py	Loads a TrackViews for given genomic region. Start and end indices are set correctly based on input region
VirtualTrackColumn.py	A wrapper for the columns of track table.
gtrackcore.track.pytables.database	
___init___.py*	A method that closes remaining open PyTables files and stores metadata. This is registered to the atexit module
Database.py*	The module where a PyTables file is managed, as described in Section 4.2.3.2
IndexRetrieval.py	Where the `TrackView` start and end indices are found
MetadataHandler.py*	Functionality related to the metadata handling. as described in Section 4.3.3.1
Queries.py	Queries used for bounding region table
gtrackcore.tools	
MergeApi.py*	Described in Section 4.6.2
gtrackcore.tools.suite	
GTrackCoreSuite.py*	The main module for the GTrackSuite that was described in Section 4.6.1

Chapter 5

Run Time Performance and Storage Efficiency

This chapter will present concrete measures of how the preprocessor of the new GTrackCore compares to preprocessor of the old GTrackCore. The results from these measures are divided into two categories, namely benchmarks in form of *average run times*, and storage efficiency in form of *combined file sizes*. The results have been obtained by running the preprocessor on different test tracks, i.e. selected tracks for the preprocessor, believed to cover most of the different execution paths that can be taken by the preprocessor.

5.1 The test environment

The instance of the Genomic HyperBrowser, in which the Preprocessor performance tool is embedded, is run on a dedicated server that has 32 physical CPU cores, each with an operating frequency of 2.3GHz, and approximately 500 Gb of RAM. The server has access to the parallel file system of the Abel Computing Cluster, named FhGFS (Fraunhofer global file system).

The server was not exclusively dedicated to our tests, and was used by others to perform analyses as well. It was experienced that the performance tool not always produced consistent results, which was likely due to this fact. One of the disturbing elements was eliminated by moving the test data to a separate partition, with less load that the one used by the main HyperBrowser installation. Additionally, the testing was tried to be done at times when the load on the server was low.

5.2 The textual genomic tracks used for testing

The 'selected test tracks' (see Section 3.4.3.1) and the various settings that the results are based on are found in Table 5.1. The selected tracks are believed to cover many of the different scenarios that can occur during preprocessing. Because both sorting has already been discussed in chapter 4, we have not actively tested this parameter here.

5.3 Results from the preprocessor

Table 5.1: The 'selected test tracks' used for testing, that together cover many of the unique scenarios that might occur during preprocessing.

Track id	Track type	# elements	# track sources	# columns	Table resized?	Write method
Sequence	Dense	3 095 677 412	24	1	No	WRS
Bendability	Dense	689 702 695	3	1	No	WE
Repeating elements	Sparse	5 232 241	1	6	No	WE
Repeating elements$_2$	Sparse	5 232 241	1	12	No	WE
Genes	Sparse	182 798	1	12	No	WE
Interconnections	Dense	3 076	23	4	Once	WE
Parkinson's	Sparse	67	1	16	No	WE

#track sources Number of textual track sources that are to be parsed and preprocessed
Table resized? Shape or itemsize is larger in a subsequent source, hence a table resized is needed
WE `writeElement`
WRS `writeRawSlice`

5.3.1 Overview

All the results presented in this section are made available in Appendix A. The results can be reproduced by using the 'Preprocessor performance tool', which is also made available there.

Table 5.2 contains results related to run times, while Table 5.3 contains results related to file sizes of the binary data. Table 5.3a and 5.2a, respectively shows run times and file sizes *compared* between the old GTrackCore and the new GTrackCore. Table 5.2b and 5.3b, respectively shows run times and file sizes of the new GTrackCore with different *Blosc compression levels* (see Section 2.6.5.2). The different Blosc compression levels that have been included are level 0/no compression (c0), level 1 (c1), level 5 (c5) and level 9 (c9).

Table 5.4 shows how large impact the 'create arrays'-option has on run times (see Section 4.3.2.3), and compares two different configurations of the new GTrackCore, i.e. with array creation activated and deactivated.

The last table (Table 5.5) shows the number of files produced by the preprocessor compared between the new GTrackCore and the old GTrackCore. The number of files have been counted manually by the bash command '`find . -maxdepth 1 -type f -not -name ".*" | wc -l`' (find all *files* of the current directory, excluding hidden ones, and then count them). The old GTrackCore has files of sparse tracks in two separate directories, i.e. `withOverlaps` and `noOverlaps`, thus two counts were in this case added.

5.3.2 Run time

The following hypotheses have been directly extracted from the run time results.

Table 5.2: Results related to run times of the preprocessor, based on test runs performed on the tracks listed in Table 5.1.

(a) Run times compared between new GTrackCore (pytables) and old GTrackCore (memmap). For rows with '% change' < 0, the new GTrackCore is more efficient

	Avg. of 10 runs (sec)		
Track id	memmap	pytables	~% change
Sequence	2 848	7 534	164.53
Bendability	25 373	25 975	2.37
Repeating elements	2 448	1 738	−29.00
Repeating elements$_2$	3 759	2 617	−30.38
Genes	204	191	−6.81
Interconnections	681	660	−3.08
Parkinson's	2.6	4.3	65.38

(b) Run times of the new GTrackCore, with different Blosc compression levels.

	Avg. of 10 runs (sec)			
Track id	c0	c1	c5	c9
Sequence	7 534	7 612	7 511	6 832
Bendability	25 975	26 164	28 193	25 989
Repeating elements	1 738	1 723	1 770	1 750
Genes	191	232	228	233
Interconnections	660	682	684	681
Parkinson's	4.3	4.6	4.4	4.3

Sparse tracks are normally faster in the new GTrackCore We see that the new GTrackCore, without compression, is faster than the old GTrackCore for the two sparse tracks 'Repeating elements' and 'Genes. Respectively 29% and 6.81% faster. The last sparse track, 'Parkinson's', however, deviates from this and is a lot slower in the new GTrackCore than in the old, i.e. 65.38% slower. In other words, for larger sparse tracks, i.e. 'Genes' and 'Repeating elements', it seems like the new GTrackCore performs better, while the old GTrackCore performs better when preprocessing smaller sparse tracks, i.e. 'Parkinson's'. Derived from this, it seems like the new GTrackCore at a certain threshold, when the raw track has enough elements, becomes faster than the old, and that the difference becomes larger as the number of elements increases.

The more columns the larger the difference The difference between 'Repeating Elements$_2$ in the new and old GTrackCore is a little larger than the difference between 'Repeating Elements' in the new and old GTrackCore, i.e. $30.38\% > 29.00\%$. These two tracks have exactly the same properties except from number of columns. 'Repeating Elements$_2$' has twice as many columns as 'Repeating elements'. Thus it seems like the new GTrackCore is somewhat better at handling more columns.

Preprocessing of single-column tracks may be slower in the new GTrackCore Preprocessing of the sparse track 'Interconnections' is about 3, 08% faster in the new GTrackCore, despite the fact that the preprocessor in this version of GTrackCore is entering the resize method once, as can be seen in the overview Table 5.1. The sparse 'Bendability' takes about is 2.37% longer time preprocess in the new GTrackCore. 'Interconnections' has 4 columns, while 'Bendability' has a single column'. This may indicate that the new GTrackCore is slower when preprocessing single-column tracks.

writeRawSlice is slower in the new GTrackCore 'Sequence' is a full 164.53% slower in the new GTrackCore compared to the old. This track is, as we can see in the overview Table 5.1, using the `writeRawSlice` method, which is appending a 'slice' of genomic elements directly to a table/memmap.

writeRawSlice is faster than writeElement 'Bendability' is, with regard to informational content, very similar to 'Sequence', i.e. both tracks have a single 'val' column. The preprocessor is, however, instead using the `writeElement` method to store genomic elements in the table. 'Bendabiliy' has approximately 6.5 times fewer elements than 'Sequence', but the processing of the former takes about 3.5 times longer in the new GTrackCore.

Adding CArrays does not have great impact Adding arrays does not seem to have very large impact on run times. In Table 5.4, we see that preprocessing time of the track 'Repeating elements' increases with only 2.78% – not much if arrays proves to be faster than tables in situations where the retriever module is slicing arrays, i.e. a `get<Column>AsNumpyArray` is done in the `TrackView`.

Adding compression does not have great impact Adding compression does not seem to have a very large impact on preprocessing time. A situation where the run time has increased a little, is for the track 'Bendability' with Blosc compression level 5, where decrease in preprocessing time is about 8% compared to compression level 0, i.e. without compression. The run time of 'Bendability ' with compression level 9, is about the same as compression level 0. The compression ratio for compression level 9 is additionally higher than for compression level 5, which makes the decrease in preprocessing time for level 5 a little surprising. An interesting thing to notice is that the track 'Repeating elements' for compression level 1 shows a decrease in preprocessing time.

5.3.3 Binary file sizes

Similar file sizes without compression The binary data sizes are similar between the two version of GTrackCore when the new GTrackCore has compression deactivated. Surprisingly, for two tracks, i.e. 'Repeating elements', 'Interconnections' the binary data has increased in sized in the new GTrackCore.

Some tracks have good compression ratio When compression in activated in the new GTrackCore one can see a decrease in data sizes for all tracks. The compression ratio is highest for the sparse tracks. The track 'Genes' has for instance a ratio 15.50 for compression level 1 and around 30 for compression level 5 and 9. The compression ratio is exclusively decided by the contents of the track, so 'Genes' must for instance, have a lot of repeating patterns that can be represented with fewer bytes. A bit surprising is that the track 'Sequence' and 'Repeating elements' grows in size at compression level 9, compared to compression level 5.

Adding CArrays doubles file size When the option to create arrays is enabled, the track expectedly grows to about double the size. This is expected as the informational content of an array is the same as for a table, i.e. each array corresponds to a table column.

5.3.4 Number of files

The number of binary files created by the preprocessor is much lower in the new GTrackCore. In fact, there is only created a single PyTables file for any track since both the with- and no-overlaps tables, the bounding region table, and the `TrackInfo.shelve` are located in the same file. The improvement is obviously largest for tracks with many columns, such as the track 'Parkinson's'. The smallest number of files in the old GTrackCore, for any track type, is two. This is because every track at least must have a single column and a `boundingRegion.shelve`.

Table 5.3: Results related to file sizes of the binary data produced by the preprocessor, based on test runs performed on the tracks listed in Table 5.1.

(a) File sizes compared between new GTrackCore (pytables) and old GTrackCore (memmap). For rows with '% change' < 0, the new GTrackCore is more efficient.

	Tot. file sizes (Mb)		
Track id	memmap	pytables	~% change
Sequence	3 095.69	3 096.00	0.01
Bendability	5 518.67	5 518.67	0.01
Repeating elements	836.37	898.35	7.41
Genes	852.10	853.78	0.19
Interconnections	188.13	203.73	8.29
Parkinson's	0.56	0.55	−1.78

(b) File sizes of the new GTrackCore, with different Blosc compression levels.

	Tot file sizes (Mb)			
Track id	c0	c1	c5	c9
Sequence	3 096.69	2 867.55	1 511.00	1 666.81
Bendability	5 518.67	5 296.41	4 781.17	4423.81
Repeating elements	898.35	396.37	167.11	175.33
Genes	853.78	55.10	27.50	26.68
Interconnections	203.73	202.10	74.56	66.20
Parkinson's	0.55	0.36	0.07	0.07

(c) Compression ratios for different Blosc compression levels.

	Compression ratio		
Track id	c1	c5	c9
Sequence	1.08	2.04	1.86
Bendability	1.04	1.15	1.25
Repeating elements	2.27	5.38	5.12
Genes	15.50	31.05	32.00
Interconnections	1.01	2.73	3.09
Parkinson's	1.52	7.86	7.86

Table 5.4: Results from the preprocessor, based on test runs performed on the track 'Repeating elements' from Table 5.1, without compression. Run times and file sizes are compared between two configurations of the new GTrackCore. One with the 'create_arrays' option enabled, i.e. 'pytables (arrays on)' and one with the the create_arrays option disabled, i.e. 'pytables (arrays off)'.

(a) Run times of the preprocessor

	Avg. of 10 runs (sec)		
Track id	pytables (arrays off)	pytables (arrays on)	~% change
Repeating elements	1 657	1 703	2.78

(b) Combined file sizes of the binary data. For rows with '% change' < 0, pytables (arrays on) is the better performing one.

	Tot. file sizes (Mb)		
Track id	pytables (arrays off)	pytables (arrays on)	~% change
Repeating elements	898.34	1 796.75	100.00

Table 5.5: The number of files produced by the preprocessor, compared between new GTrackCore (pytables) and old GTrackCore (memmap).

	Number of files	
Track id	memmap	pytables
Sequence	2	1
Bendability	2	1
Repeating elements	16	1
Genes	28	1
Interconnections	7	1
Parkinson's	40	1

Chapter 6

Discussion

6.1 Analysis and discussion of results

In this section there will be done an analysis based on the results related to performance and storage efficiency that were presented in chapter 5. The new GTrackCore is set up against the old GTrackCore, in an attempt to determine how the PyTables-based data model compares to the existing memmap-based model. Included in the assessment is an examination of how well the new model performs compared to the old, and how much impact compression has on both run times and file sizes.

The analysis of the run time results will be supported by profiles, which are made available in Appendix A, that may give an answer to why the results are as they are with regard to the time spent in different methods of the preprocessor. Because of the overhead and the fact that the profiles are based on single runs they will not give definitive answers. They will, however, give a good indication of how GTrackCore performs at method level.

Faster preprocessing of sparse tracks

That the sparse tracks 'Genes' and 'Repeating elements' are faster to preprocess in the new GTrackCore may have something to do with sparse tracks having genome elements extracted from the track created in the with-overlaps phase where overlaps removed by doing a clustering. Preprocessing profiles of the track 'Genes' for both the new and old GTrackCore point towards this, i.e. that the call tree growing from the module where the clustering happens actually is the source of the speedup. In the new GTrackCore, the time spent in the iterator method of the clusterer, i.e. `GEOverlapClustererBase`, cumulates to 163 seconds. In the old GTrackCore the time spent in the same method is 191 seconds, indicating an improvement. Profiles of the larger track 'Repeating elements' shows an even larger increase: In the new GTrackCore, the time spent in the method accumulates to 828 seconds, while in the old GTrackCore it accumulates to 1471 seconds. The speedup can be explained by the fact that the iterator of the GTrackCore yields `PytablesTrackElements` while the iterator of the old GTrackCore yields `TrackElements`. This explanation is supported by

the results that concerns retrieval, that clearly shows that the iterator in fact is faster in the new GTrackCore [38].

The sparse track 'Parkinson's' breaks with the tendency of the new GTrackCore being able to cluster faster than the old GTrackCore. A plausible explanation for the discrepancy is that 'Parkinson's' is very small in comparison with the two other tracks and to a greater degree is affected by the overhead of creating a `TrackView`. This is underlined by the profiles. In the new GTrackCore, the method where the `TrackView` is loaded cumulates to 2.30 seconds. In the old GTrackCore the `TrackView` is loaded in 0.319 seconds. The reason for why this is so much slower by use of PyTables is primarily because of overhead associated with use of internal PyTables methods. For example, the `_get_node` method of PyTables is relatively slow and is called several times when the preprocessor loads a `TrackView`.

Performance problems related to appending slices

As the result from the preprocessing of the track 'Sequence' indicates, the `writeRawSlice` method is considerably slower in the new GTrackCore. A plausible explanation for the decrease in performance is the fact that there is more overhead in the process of PyTables calling `append`, than it is assigning a NumPy ndarray to an existing memmap. In the old GTrackCore, memmaps large enough to contain the entire track are created upon initialisation of the `OutputManager`. This means that `writeRawSlice` simply has to do a slice assignment, where contiguous positions of an already existing ndarray are overwritten. In the new GTrackCore, the same operation is more complicated, due to the fact that HDF5 is full-featured and a far more sophisticated format than NumPy memmaps, and involves to convert ndarrays to either a new or existing HDF5 chunk that in turn has to be written. Hence, when `append` of PyTables is called many times, e.g. the profile of 'Sequence' shows 619 113 560 calls, the overhead of the underlying HDF5 operation, which is larger than the small overhead of NumPy assigning an ndarray to a slice of a memmap, accumulates and influences the run time significantly.

The previous is confirmed by profiles of 'Sequence'. The cumulative time spent in `_add_slice_element_as_rows`, which appends slices to a table in the new GTrackCore, is 7 017 seconds. 269 of these seconds are local, i.e. 'tottime' spent inside its own method body, thus the majority of remaining $7017 - 269 = 6748$ seconds must have been spent within the `append` method of PyTables. The cumulative time spent in the corresponding method of the old GTrackCore, where slices are assigned to a NumPy memmap, is 471 seconds. 398 of these are local. The remaining $471 - 398 = 73$ seconds must have been spent in the underlying NumPy method. This proves that the overhead of writing slices in PyTables is much greater than that of NumPy.

While the `writeRawSlice` method is slower in the new GTrackCore than in the old, the benchmarks strongly suggest that it is far more efficient than the `writeElement` method, in situations where both can be used. Because 'Bendability' has a single 'val' column it could have used `writeRawSlice` as well, since the track has a single 'val' column, like 'Sequence'. This

would very likely have made the track faster to preprocess, also in the new GTrackCore. Even though it would have meant an even larger speedup in the old GTrackCore. In order to enable this, the workings of the parser would have to be changed so that textual track formats other than the FASTA format could have been read 'slices' at a time.

Number of columns may affect run time

If the number of columns affects run times, in favour of the new GTrackCore, the difference between the two versions in terms of cumulative time spent in the `writeElement` method of 'Repeating elements$_2$' (which is the same track as 'Repeating elements' except that it has more columns. See Section 3.4.3.1) should be larger than the difference between the cumulative time spent in the same method of the 'Repeating elements'. For 'Repeating elements$_2$', the `writeElement` method of the new GTrackCore uses 245 seconds, while the corresponding method of the old uses 986 seconds. The difference here is $986 - 245 = 741$. For 'Repeating Elements', the `writeElement` method of the new GTrackCore uses 180 seconds, while the corresponding method of the old uses 412 seconds. The difference is $412 - 180 = 256$. The difference is, as we can see, larger for 'Repeating Elements$_2$ which indicates that the new GTrackCore is more scalable than the old GTrackCore, with regard to the number of columns. In fact, it seems like the new GTrackCore write elements $986/245 \approx 4$ times faster than the old GTrackCore. Even though the benchmark results shows a total speedup of only 1.38% percentage points.

There can be several reasons why the new GTrackCore apparently is better at this. First of all, It should be noted that the composition of classes involved in storage of genomic elements is more complex in the old GTrackCore (mentioned at the beginning Section 4.4), due to that each column is stored in a separate file: The `OutputManager` has a `OutputDirectory` (the leftIndex and the rightIndex are made here), that again has several `OutputFile` objects. While all these objects will add some overhead that can become apparent when preprocessing large enough track, the seemingly most time-consuming factor is that the old GTrackCore has to write elements to several files. For 'Repeating elements$_2$' for instance, the underlying method of the old GTrackCore, that adds elements to memmaps, is called 113 763 793 times, accumulating to 489 seconds. The `_add_ge_dict_as_row` of the new GTrackCoreis, on the other hand, called 10 342 163 times,[1] accumulating to 230 seconds. The new GTrackCore does in other words enter the underlying write method once for each genome elements that is to be written, while the old GTrackCore enters its corresponding method |columns| times. According to the profile, the time spent within the write method (inside its method body, not in submethods) is lower in the old GTrackCore, i.e. 0.00000 seconds (lower than the clock tick rate) vs. 0.00002 of the new, but because more files are written, the old GTrackCore

[1] Note that since both with and no-overlaps tables are created, the number of elements written are about double in size compared to the number of elements in the text files. The reason why it is not exactly double in size is because some of the elements are clustered

is slower than the new.

That the old GTrackCore has to write elements to N different column files may have an impact on large tracks with many columns. It is conceivable that if the number of elements and columns is large enough, the difference between new and old GTrackCore could be even greater: For example, a track with 10 times as many genomic elements as 'Repeating elements$_2$ and with 20 columns/fields instead of 12, that has no overlapping elements, would have had $2 \times 10 \times 52\,322\,410 = 104\,644\,820$ elements to write (no overlapping elements means that the same data is stored twice in both the with and no-overlap binary tracks). In the old GTrackCore this would take $104\,644\,820 \times 20$ write operations since each column-file has to be written once for every element. Based on the previously presented profiles of 'Repeating elements$_2$, for the old GTrackCore, this would use $\frac{104\,644\,820 \times 20}{113\,763\,793} \times 489 \approx 8996$ seconds. In the new GTrackCore the same would use $(104\,644\,820/10\,342\,163) \times 230 \approx 2327$ seconds. By this we see that, even though the difference would become greater, it would likely not affect total running times that much. As we mentioned, the benchmark results from runs of 'Repeating elements' and 'Repeating elements$_2$ only shows a total speedup of a mere 1.38% percentage points. This suggests that the time spent in the write method is negligible, and the difference between the two versions would come apparent only for extremely large tracks with a large number of columns. On the other hand, it is possible that the overhead of the old GTrackCore then again would have had an even greater impact, perhaps especially the creation of the leftIndex and right Index, which could have made the new GTrackCore excel, but more testing is required in order to confirm this.

Small difference in preprocessing time of single-column sparse tracks

The hypothesis that writing single columns is slower in the new GTrackCore is not confirmed by the profiles. Although the benchmark shows that preprocessing of 'Bendability' is somewhat slower in the new GTrackCore, the profiles suggest the opposite. For instance, the `writeElement` of the new GTrackCore accumulates to 7 157 seconds, while the corresponding, outer, method in the old GTrackCore accumulates to 7 598 seconds. This is not consistent with the fact that the time spent in the method that actually writes elements is lower in the old GTrackCore, i. e. 1 157 in the old and 6 306 in the new. This must mean that it is the overhead object creation of the old GTrackCore that is the cause of the small difference between the two versions that shows up in the profiles.

We have not found a plausible explanation for why the new GTrackCore in this case, according to the benchmarks, performs worse than the old. Nothing in the profiles indicates this. However, it seems reasonable to conclude that the difference between the two versions, when processing sparse single column tracks like 'Bendability', is too small to be of any practical significance.

Implications of adding compression

From a storage perspective, adding compression is undoubtedly beneficial: Most of the test tracks shows good compression ratios, even at low Blosc compression levels, and the increase in preprocessing time is insignificant. However, there is no getting away from the fact that the retriever module is more performance critical than the preprocessor; retrieval is done an indefinite number of times, whereas preprocessing normally only is done once. Hence, for compression to really be considered, the retriever has to perform satisfactory with it added.

According to the results related to retrieval, the retriever module performs consistently *worse* when compression is added, especially at compression level 5 and 9. The retrieval results indicate that the `asNumpyArray` table-slicing is affected more than the iterator. However when using *arrays* instead of tables, the results for the `asNumpyArray` retrieval method, with and without compression, are somewhat similar and in some cases actually faster in the new GTrackCore. [38] This seems to be because how `Arrays` are consisting of elements of atomic type instead of elements of composite type such as `Tables`. We will come back to this in Section 6.2.

Considering that the results regarding retrieval of compressed data generally are negative, the big question is whether one is willing to give up some retrieval performance for smaller binary data sizes. If GTrackCore were to be loosely coupled with the main installation HyperBrowser, which has virtually unlimited storage space, the answer to this question would probably be a resounding no. On the other hand, if GTrackCore were to be used together with a lightweight command-line based toolset, developed for use on lower-end private computers, it could in some cases be appropriate to sacrifice some performance for smaller data sizes. Many genomic tracks are large, and personal machines do not normally have unlimited storage.

Summary

Sparse tracks are generally faster to preprocess in the new GTrackCore, due to the iterator being faster. Only very small tracks are slower because the overhead of creating a `TrackView` is larger than in the old GTrackCore.

The `writeRawSlice` method, used to write many elements to file in one go, is considerably slower in the new GTrackCore, but it is still faster than writing elements through `writeElement` for tracks that in theory can use the former method.

Tracks with many columns are seemingly faster to preprocess in the new GTrackCore, because of less overhead and because elements in the old GTrackCore has to be written to multiple files instead of one.

Dense tracks with a single column takes about the same time to preprocess in the two versions. While the new GTrackCore according to the benchmark results is somewhat slower than the old, the profiles suggest otherwise. The exact reason why benchmarks shows that the new GTrackCore is slower has not been found, although it is believed to have something to do with difference in load on server where the tests were run.

The results from adding compression are positive from a storage perspective, but compression has a negative impact on retrieval times. As retrieval speed normally is more important than file sizes, compression should probably not be added unless experimenting with the chunkshape variable gives positive results.

6.1.1 Weaknesses in analysis

During the analysis of the track 'Interconnections' it was discovered that it does not enter the 'resize' method enough times for it to be thoroughly tested – it is in fact only entered once. Despite that the benchmark results of the tracks shows a positive decrease in run times, i.e. the opposite of what was expected, it is believed that the resize operation will have noticeable negative impact on the preprocessing time of very large tracks. Nevertheless, the fact that we not were able to immediately find any such tracks may indicate that they are not common and that this is a rare scenario that is not very important.

The cProfile profiler does not very detailed information about C extensions, and only the first-level of calls is timed. Thus, the analysis could not give a detailed explanation of exactly why underlying methods performed as they did. The call hierarchy could for instance have given interesting information about the workings of PyTables and HDF5, although this information naturally can be found by analysing source code, as both these projects are open source.

In order to make the run time results more precise, they have been averaged over 10 runs. While a larger number of runs would have been preferable, one of the test tracks (Bendability) took too long to process for it to be feasible. However, although some of the runs that the results are based on had some outliers, none of them substantially affected the averages.

6.2 Using heterogeneous tables instead of homogeneous arrays

When the work of incorporating PyTables in GTrackCore began, we were convinced that the `Table` was the data container to use. It was for instance believed that the query functionality of PyTables was the central feature that really made the package unique and would make the new GTrackCore outperform the old – likely due to the fact that the feature is extensively advertised at the front page of the PyTables site. However, table queries proved to be inefficient at the places where we though they would lead to a speedup, e.g. finding something known as 'start' and 'end' indices when loading a `TrackView`. Creating an index for the table did not work either, and profiles suggested that the `TrackView` loading was a big bottleneck that had to be optimised. Hence we instead went with a manual way of finding these 'start' and 'end' indices, abandoning what was initially believed to be the killer feature of PyTables.

When it came to this we were not using the arguably most important

feature of `Tables`, and could practically have switched to using `Arrays` instead. However, without knowing exactly how `Arrays` would perform compared to `Tables`, it was decided to make it optional to create them in the local finalisation step, as we described in Section 4.3.2.3, so that retrieval could be tested with these as well. The reason why it was believed that use of `Arrays` could increase retrieval speeds, was because less data likely would have to be loaded in cases where track columns 'sliced', i.e. a part of the column is read into memory. A `Table` is, as we remember from Section 2.6.4.2, a HDF5 data set with elements of a *compound* data type, which on disk basically are represented as C structs stored one after another. Hence, when a single column of a `Table` is sliced, the contents of all the other fields have to be loaded as well. `Arrays`, on the other hand, have elements of *atomic* type. This means that a slice operation performed on `Arrays` do not require loading of unnecessary fields, something that in theory should make `Arrays` faster than `Tables` when data is retrieved by slicing columns.

The performance tests done by Skifjeld [38] indicates that `Arrays` in fact are faster than `Tables` when doing 'as NumPy array'-slicing, which might be a consequence of the aforementioned, i.e. that arrays are structured in a manner that more suitable for slicing. It should be noted he iterator-based retrieval method has not been tested together with arrays, but is suspected that this is somewhat slower with arrays since several arrays would have to be traversed instead of a single table. If this proves to be the case, a possible solution can be to use `Arrays` for slicing and `Tables` for iteration, although this would double the space needed to store the genomic tracks. From a storage perspective there is little difference between `Tables` and `Arrays` since they take up approximately the same space. If anything, `Tables` are a bit more practical since related data is collected in the same data set, which for instance makes it easier to view and edit data later.

We acknowledge the fact that we during this project have not spent much time experimenting with the more advanced concepts of PyTables and the HDF5 format, as the main objective, which was to incorporate PyTables in GTrackCore, was too time-consuming. For example, the `chunkshape` variable, that currently is set automatically through the `expectedrows` variable upon table creation (see Section 4.4.1.2), could perhaps have been investigated further to see whether it could have improved retrieval performance. The 'Optimization tips' provided in the PyTables User's guide [33] encourage to experiment with the chunkshape variable when one has 'special requirements' for how retrieval should be done – and GTrackCore certainly have this. Skifjeld [38] presents some simple experiments related to retrieving data when the chunkshape have been set manually. The experiments suggest that it is difficult to do something sensible with it because of the many ways track data can be retrieved, and that the default `expectedrows`-based chunkshape likely is close to optimal because of this. However, it is possible that a more sophisticated procedure that before the table is created calculates the chunkshape based on the analyses that are expected to be performed on the track, e.g. how many and how large slices that are typically retrieved, is something that is worth experimenting more with.

6.3 Is the new GTrackCore reliable?

The test environment that comes with the GTrackCore package gives valuable information about how the package is supposed to work. This was utilised during the development of the new GTrackCore. New functionality was added, and existing functionality was refactored, based on failing tests. If something did not work properly, direct feedback was received about it and we could immediately try to repair it. While this helped a great deal, it was not always easy to identify exactly what to repair. This was due to the integration tests asserting that output from the retriever corresponds to some hardcoded values, e.g. that the number of elements retrieved from a certain Segments track of a test case should be 3. When tests like this failed, it was difficult to know whether it was caused by the preprocessor or the retriever module, forcing us to engage in complex manual debugging sessions.

There were also some problems related to the fact that important functionality was not covered by any tests. During the development, a passing test case was taken taken that the feature that the test covered was working. Hence, when all the tests passed it was taken as that the new GTrackCore had become fully functional. However, during the creation of various track operation-tools used to test performance of the retriever module, it was discovered that this was not the case. A few of the newly created operation tools, which relied on traversing Points tracks, produced results that differed between the two versions of GTrackCore. It was established that the old GTrackCore was the one producing correct results, and that the source of the problems was the iterator of new GTrackCore yielding elements with incorrect 'start' attributes. The problem proved to be related to attributes not being relative to something known as a 'genome anchor' – a concept which is explained by Skifjeld [38]. Although being seemingly important functionality, it was not being tested by the tests that came with GTrackCore.

While we were able to find and repair this exact bug, there is chance that there might be other cases that have not been caught by any of the supplied tests. Thus, to whether the new GTrackCore is 100 % reliable, the answer is probably a 'no'. Nevertheless, it is safe to say that without the test environment the development process would have taken much longer time and that the final code product would have been a considerably less reliable than what it is today. Even though passing tests are not a guarantee that everything works, it is at least a good indication of quality.

6.3.1 Creating missing test cases

An example of functionality directly relevant to the preprocessor that was not tested properly, was whether subsequent `OutputManagers` are updated with correct shapes and itemsizes (see Section 4.4.1). Because there were no test cases for this situation, it took some time to realise why the system threw errors seemingly at random. Immediately, after the requirement was understood, the '`TestTrackPreProcessorTwoFiles`' class was added

in the `test/preprocess` package. The tests in this class are run upon two different test cases (see Section 3.3.1.1), with respective track types Linked Valued Segments and Linked Genome Partition. Both of these test cases are consisting of two files, where the first file has smaller shape and itemsize than the second. The single assertion that is done is to check that the preprocessed data that is based on the tests case, is equal to the test case itself.

6.4 The research questions

6.4.1 Is PyTables suitable for storage of genomic tracks in GTrackCore?

Is PyTables, and the underlying HDF5 format, suitable for storage of genomic tracks in GTrackCore, and will PyTables solve the problems related to multiple files being needed to represent each individual data set?

The implementation presented in Chapter 4, together with the measurements of performance and storage efficiency from Chapter 5, strongly suggests that PyTables and the underlying HDF5 format is suitable for storage genomic tracks in GTrackCore.

The run time results show that the preprocessor of the new PyTables-based GTrackCore most of the time either performs approximately the same or better that the memmap-based GTrackCore, e.g. preprocessing of sparse tracks. In some cases, however, PyTables performs significantly worse, e.g. writing slices through `writeRawSlice`. Nonetheless, most track types does not even support use of `writeRawSlice`, and extended use of it would have required more development for both versions of GTrackCore. That being said, performance is not the most important aspect of the preprocessor, since it essentially is run once for each track. High performance in the retriever module is definitely the more important, and since PyTables has shown satisfactory and good results in many other situations it is likely accepted that preprocessing of this exact track type is slower when using PyTables.

Utilisation of PyTables solves the problem of the ad-hoc data model's extensive use of files to represent genomic tracks by having all columns of a track stored in a single table, which again are stored in a single file that has an internal node structure that replicates the old directory structure. Through the creation of the `mergeAPI`, which was presented in Section 4.6.2, it has additionally been shown that it is highly possible to have multiple tracks stored in the same PyTables file. This can mean much with regard to practicality, since it for instance can be used to add semantics. For example, having two tracks bundled in the same PyTables file can be used explicitly express that tracks are related and are intended to be used in similar analysis. The underlying HDF5 format of PyTables also makes it more convenient to share tracks since only a single file has to be dealt with, and it does for instance solve the problem of Galaxy's lacking support for multi-file upload.

6.4.2 Can PyTables be used in GTrackCore as it stands, or should GTrackCore be refactored?

Can PyTables be used for storage of genomic data in GTrackCore as it stands, or should GTrackCore be adapted to it through a comprehensive code restructuring?

The implementation of a PyTables-based GTrackCore was carried out without dramatic changes to the existing source code. Hence the answer to whether PyTables *can* be used with GTrackCore as it stands is probably a yes. However, during the development it was discovered that GTrackCore in many cases probably would have benefited from being built from the ground up for use with PyTables.

One problem with GTrackCore's current design is that the textual tracks are parsed and stored as Python attributes in instances of the `GenomeElement` class, which later has to be converted to HDF5 data structures. This is a slow process, and it would be more efficient to skip the step of using Python's built-in data structures, and instead use NumPy arrays directly. NumPy types are more similar to the HDF5 types than what the Python types are. In fact, in some cases they are exactly the same. Less use of the slow Python interpreter will generally increase performance. Hence, a relatively big improvement would have been to increasingly exploit the fact that the elements are to be written as rows to PyTables tables.

An example of a refactoring that can make PyTables an even better fit for GTrackCore is presented in Chapter 7, along with other suggestions for future work.

6.4.3 What are the advantages of using PyTables and HDF5 to store genomic tracks?

What are the advantages and disadvantages of using a popular package such as PyTables, which is built upon the de facto HDF5 format, for storage of genomic data, compared to using an ad-hoc format such as the custom memmap-based format created for GTrackCore?

The old GTrackCore is based on use of NumPy memmaps and there are, as we have seen, problems related to this binary data format, e.g. excessive use of files to represent a single data set. In this project we have tried to address these problems by incorporating PyTables in the GTrackCore package. However, it is conceivable that these problems could have been solved by a far simpler solution than the major refactoring that the incorporation of PyTables has involved – for example by putting the memmaps inside some kind of container so that the current use of them could have been continued. Therefore, for PyTables to really be considered a sensible replacement it has to bring something unique to the table that NumPy and the memmap-based format lacks.

There are in fact several advantages to using PyTables, and a few of these have already been mentioned. For example, the ability of PyTables to easily add compression, and its ability to store several data sets in a single file. These examples are, however, both consequences of the fact that

PyTables is an abstraction level on top of HDF5, which again probably is an advantage in itself.

The HDF5 format is highly portable and has wide support across platforms and programming languages. This is advantageous since it means that binary tracks created through the GTrackCore package, with use of PyTables, do not have to be managed exclusively by Python and NumPy. For example, there are HDF5 interfaces to both the popular Java language [26] and R [34], which is widely used in statistical software. Since HDF5 is not built around a single integrated data model, which the memmap-based format certainly is, it enables users to create their own tools that may use the binary tracks in other ways than the two supported retrieval scenarios of GTrackCore, i.e. 'column as array'-extraction and iteration. An example of such a tool may for instance be one that performs queries on the tracks, through the high-performance query engine of PyTables.

The same cannot be said for the memmap-based format, which is not very portable and cannot be used to its full potential outside of GTrackCore. The data components of the format, i.e. the column memmaps, may be used by other Python programs, but use of the surrounding `leftIndex` and `rightIndex` structures are too tightly integrated in GTrackCore to really be utilised elsewhere. While the data components of the format are not directly tied to GTrackCore, there is nothing that makes them particularly appealing to use in other applications since they are pretty much a collection of NumPy memmaps without any outstanding features except the fact that when read into memory they can be used any other NumPy array.

Another advantage of HDF5 is that it is easier to convince others that the open-source format is flexible and supports high performance retrieval than doing the same with use of a custom ad-hoc format, which from a reputation perspective is important if GTrackCore is to be used together with a command-line based analysis toolset. HDF5 has been in development for over 20 years, and has likely far more features than any ad-hoc format can advertise with. In addition, the HDF Group that has created the format is committed to ensuring that HDF5-stored data is accessible a long time into the future [44]. Since HDF5 is a popular library and format, there are several tools that make it easier to handle and manage data. An example of this is how there are several GUI tools for browsing HDF5 data graphically. These make it possible to visually interpret the data before, say, running an analysis job on it. Even though this is possible to do with the memmap-based format as well, it is much more tedious since it would require to either take a look at the data through a Python shell programmatically, or develop an equivalent GUI tool.

Furthermore, an ad-hoc format needs continuous maintenance and improvement. This consumes development resources that could have been used on the primary objective of the project, which in the case of the Genomic HyperBrowser arguably is to make reliable and efficient analysis tools that are accessible to non-technical users. PyTables and HDF5 does for instance not need the use of complex external index structures such as the ones created for the memmap-based format of GTrackCore, which is positive since it is one less module that would require maintenance.

Above are some of the advantages of using PyTables and HDF5 to store genomic data, but there are disadvantages as well. Especially when it comes to use of PyTables in the GTrackCore package – at least as the package stands today. GTrackCore has been designed for the ground up for efficient use of Numpy memmaps. The model and the way it handles the memmaps is specifically designed for how the analysis tools currently operate. To simply integrate PyTables in a system that has been built around efficient use of NumPy memmaps is unlikely going to be able to fully exploit the advantages of PyTables along with HDF5. The query functionality of PyTables is for instance a feature that generally is considered to be a key to high performance, but it proved to be inappropriate to use with the current system. This does not necessarily mean that the query engine of PyTables is totally unsuitable for use with the analysis surroundings, but rather that the retrieval interface that the GTrackCore package provides is too tailored for use of NumPy memmaps. HDF5 is for instance made for fast sequential processing of entire data sets, which is exactly what is done when performing analyses. Hence, it is a good possibility that a greater utilisation of PyTables and HDF5 strengths is the key to an even better performing GTrackCore and HyperBrowser.

Although the results from the test runs of the preprocessor indicates that PyTables and HDF5 is suitable for *storage* of genomic data in GTrackCore, the retrieval results are not all positive. The PyTables-based retriever performs better in some cases, but it sometimes performs worse [38]. Hence the big question is whether the cases that it performs better in, as for instance the iterator that have also affected the preprocessor positively, is recognised as being more important than the few cases in which it performs somewhat worse. Nevertheless, if it is concluded that PyTables and HDF5 are not able to compete with use of NumPy memmaps when it comes to retrieval – arguably the component of GTrackCore where high execution speed really counts – it would probably be better to look for containers that can pack several memmaps into a single file. For usage in an integrated system such as the Genomic HyperBrowser, the previously given arguments about practicality, e.g. that the HDF5 format is portable and maintained externally, are not as strong as they are for use with a command-line based toolset that were to be installed on personal computers. It is easier to get away with having a quirky format that is not easily handled in an integrated system, since its users most of the time never sees anything to it, and does not have to deal with it. That being said, if GTrackCore is to be used with a command-line based toolset, PyTables and HDF5 could be a reasonable choice, even if not being able to outperform use of NumPy memmaps. The format is more versatile, more portable than NumPy memmaps, and has a reputation for being the ultimate format for fast sequential retrieval of large and complex data.

6.5 Conclusion

In this thesis we have presented a proof-of-concept implementation of a PyTables-based preprocessor, designated as the new GTrackCore. The implementation strongly suggests that PyTables is suitable for storage of genomic data in GTrackCore, although the GTrackCore optimally could have been adapted to PyTables to a greater extent. PyTables and HDF5 solve the problems with multiple files being required to represent individual data sets, as well as the associated problems with distribution of files.

Using a renowned package such as PyTables that is built upon the feature-rich and recognised HDF5 library, which is well known for its ability to handle extremely large and complex data, makes it easier to convince other researchers that GTrackCore and the core functionality of the Genomic HyperBrowser performs well.

This project has laid the foundations for further experimentation with PyTables in GTrackCore so that the strengths of PyTables and the HDF5 format can be fully utilised. This can in turn make the tools that are integrated in the Genomic HyperBrowser, as well as the scheduled command-line based toolset, perform better.

Chapter 7

Future Work

7.1 Adapt the parser for storage of PyTables data

The parser of GTrackCore could have been rebuilt in a manner that required less work in the write methods of the `OutputManager`, by exploiting the fact that the elements are to written to a HDF5 dataset through PyTables . If the parser had read several genomic elements from the original track sources at a time, and stored these in a structure that represented blocks of table rows they could later have been appended directly through the `table.append` method (or `carray.append` if further testing proves that arrays actually are a better suited for GTrackCore). The current `GenomeElements` class could have been replaced by a '`GenomicElementBlock`' where each attribute contained a NumPy structured array [39] instead of several attributes. For example, elements with a shape could in that case have had to be inserted into an array with correct shape (as demonstrated in Listing 4.1), as soon as they were read in, which means that this functionality could have been moved from the `OutputManager`. This approach would be very similar to how `writeRawSlice` works today, but would have been applicable to any track type, not only single-column Function tracks. This could have the potential to reduce preprocessing times considerably as the results presented in Chapter 5 show that `writeRawSlice` method is about 3.5 times faster than the regular `writeElement`.

Creating this is not all straightforward, as it would require large structural changes to the entire `GenomeElement`→`GESource`→`GESourceManager`-composition. To be able to store several elements in the same NumPy array one would have to know what kind data type, shape and itemsize to use, before any elements are read. Since it for instance cannot be guaranteed that a data type with a larger shape than the current will not appear further down in the source files, this would have to calculated in advance, by iterating all the elements of the files in an extra pass, similarly to how statistics is calculated in the `_calcStatisticsInExtraPass` method. A possibility is to first use the `GenomeElements` to calculate statistics, and then, when the genomic elements of the textual genomic tracks are traversed the second time, instead use the suggested `GenomicElementBlock` that has datatype, shape, itemsize, etc., set based on the statistics retrieved from the first 'get

statistics'-pass.

If something along the lines of this was to be done, it would probably be a good idea to also change the parser so that all the original files are parsed in the same pass. As we have seen, the current way of handling one file at a time, complicates the insertion process, as a resize operation has to be done every time a new source is to be appended to the table (or possibly array).

7.2 Implement an external sorting algorithm

Because of the problems related to memory usage for both of the sorting methods that were presented in Section 4.5, it would perhaps be worthwhile to spend time on developing a custom *external* sorting algorithm that could handle tracks very large tracks, e.g. a track with 3 000 000 000 elements where one of the column data types is a 100 byte string. External sorting algorithms are meant to be used in cases where the data is too large to fit in memory, something that certainly is conceivable when preprocessing of genome-scale data sets is done on low-end machines. The external sorting method could for example have done a two-pass merge sort and have sorted chunks small enough to fit in memory, and then merged these chunks until all are sorted. Knuth [22] explains how the external sorting could have been conducted in Volume 3 of his magnum opus 'The Art of Computer Programming'.

7.3 Add GenomeInfo to the PyTables object tree

The `GenomeInfo` structure has only been mentioned briefly. The structure is quite similar to the `TrackInfo`, but instead of containing track metadata it contains metadata about the genome, e.g. all available chromosomes of the genome. The reason for why it has not been described in detail is because the `GenomeInfo` the current version of GTrackCore has been simplified with info about the genome hardcoded into its class body. In reality it is intended to behave similarly to the `TrackInfo` and be pickled in a Python shelf structure, and this is how it works in the Genomic HyperBrowser. To also have it in the common object tree of the PyTables file, something similar to how the `TrackInfo` is handled has to be done. The suggested branch to put it in is in the genome-node in the second level of the object tree, .e.g. '`/hg19/GenomeInfo`'.

7.4 Contribute to development of column-wise tables in PyTables

As we mentioned in Section 6.2, results relevant to retrieval show that 'as numpy array'-slicing is generally slower in the PyTables-based GTrackCore, which probably is due to tables being stored row-wise, i.e. every field of a each has to be loaded into memory for every operation, much alike NumPy structured arrays.

The PyTables development team is aware that a column-wise table is a thing a few people would want, and there has been made a proposal for an implementation of it [32]. They are looking for people that can help implement this.

A column-wise table could mean much in terms of performance when columns are sliced. In addition it would likely lead to better compression ratios since data stored in the same column is often more similar than data in different rows. Additionally, a contribution to the PyTables package would result in greater knowledge of the package and HDF5 – knowledge that can be utilised in the work of adapting GTrackCore to utilise PyTables to a greater extent.

Appendices

Appendix A

Results from the Preprocessor Performance Tool

The Preprocessor performance tool, that is presented in Section 3.4.3, is available at:

https://hyperbrowser.uio.no/gtrackcore/

Profiles and results from various runs done on the 'selected test tracks' can be found on the following HyperBrowser/Galaxy Page:

https://hyperbrowser.uio.no/gtrackcore/u/brynjagr/p/preprocessor-performance-tool-storage

Appendix B

Our Fork of the GTrackCore Repository

Our fork of the GTrackCore repository is located on GitHub, and can be accessed via the following link:

https://github.com/brynjagr/gtrackcore

Our main contributions have been made in the *pytables* branch.

Appendix C

Major Refactoring of the Database Module

Due to its complexity, the GTrackCore code base was not fully understood when we started coding. This added to the fact that we practically had to learn a new programming language, along with the PyTables package, made it hard to produce easily maintainable code at the first attempt. Hence, as more experience was gained, with both the code base and the language, it was discovered that some earlier design choices were the source of a complex and tangled control structure. Although the code was working, it was too hard to get an immediate grip on the control flow, especially in the `OutputManager` (see Section 4.4).

The source of these problems proved to be the *old version* of the `Database` module, or `DatabaseHandler` as it was named before (the new is presented in Section 4.2.3.2). The old version is depicted in Figure C.1.

Initially it seemed to be a good idea to have the module handle everything related to use of PyTables. That is, both management of the internal object tree and handling of tables. At this point the main `DatabaseHandler` class contained and managed both a PyTables `File` and a `Table` object. The module was designed so that the main class had multiple abstract subclass that were responsible for opening the database, i.e. the PyTables file, in different modes. The actual instantiable classes were located in one lower level of the class hierarchy.

The design problems of the module began to appear when there was added support for sorting, and for preprocessing of textual genomic track split in to multiple files, with subsequent files having shapes or itemsizses smaller than the current (see Section 4.4.1.3). According to the design, everything related to use of PyTables had to occur inside of the `database` module. Thus also copying, which both of these preprocessing features relied on. Copying content from one node to another involves creating a new table and to copy the contents of the old into it. This interfered with how these database objects had been dealt with so far, where each object was associated with a single table. The copy operation relied on having two table objects, which meant that the whole structure had to be adapted to this action. From the outside, the module became cluttered, and was difficult to use.

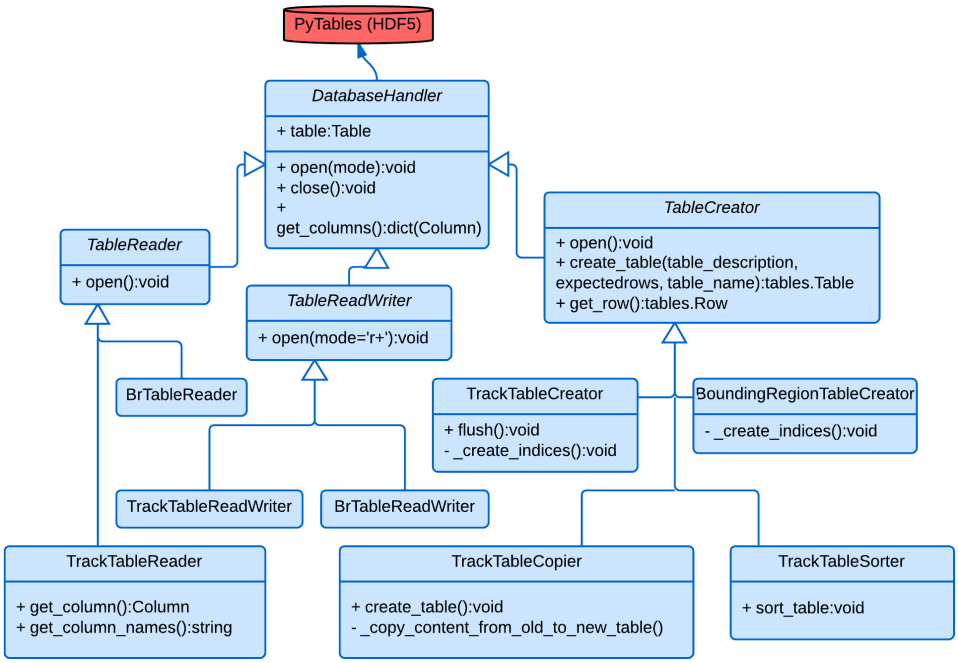


Figure C.1: How the `Database` module was structured before the major refactoring. The classes with italicised names are abstract. The structure is a lot more complex than the refactored result, which can be found in Figure 4.6 in Chapter 4.

When the design `database` module was identified to be the definite source of the mentioned problems, a major refactoring was commenced. The refactoring resulted in 118 fewer lines of code, and a much more manageable code base, that was both easier to understand and more convenient to add new functionality to. The overall code structure became flatter, and redundant classes, that only functioned as method containers, were removed. One could say that the 'You aren't gonna need it'-principle of the extreme programming (XP) development methodology was learned the hard way. Which in short terms means that functionality should not be implement things before you need them, since you can not foresee that they will be useful in the future.[1]

[1] http://www.xprogramming.com/Practices/PracNotNeed.html

Appendix D

Issue Reported on GitHub Regarding Column Assignments

The original post can be found here:
https://github.com/PyTables/PyTables/issues/338
The contents of the issue report:

I (and @brynjagr) have an issue concerning `__setitem__` in the `tables.Column` class, more precisely the `modify_column` and `modify_columns` methods in the same class.

When I try to do a slice assignment I get a ValueError saying:

```
ValueError: could not broadcast input array from shape (x,y) into shape
    (x,y,z)
```

This happens because one of the dimensions in the shape is single-dimensional, and in `modify_column` you, as you put it yourself, 'get rid of single-dimensional dimensions' when you modify the column variable with `column = column.squeeze()` (line 2406 and 2505 for `modify_column` and `modify_columns`, respectivly).

This is not the same behaviour as when using numpy ndarrays, as one might expect. Underneath is an example of how numpy handles a situation where one of the dimensions are single-dimensional.

```
In [1]: import numpy
In [2]: a = numpy.zeros((5,3,1))
In [3]: a[:] = a[:]
In [4]: a.shape
Out[4]: (5, 3, 1)
```

Here is a small example of how pytables behaves in an equivalent situation:

```
import tables
import numpy

with tables.open_file('test.h5', mode='w') as h5_file:
    table = h5_file.create_table('/', 'test', {'data':
        tables.Int32Col(shape=(3,1))})

    row = table.row
    for data in xrange(5):
        row['data'] = numpy.zeros((3, 1))
        row.append()
    table.flush()

    column = table.cols.data
    print 'column shape:', column.shape

    column[:] = column[:]
```

The example above gives this output:

```
column shape: (5, 3, 1)
Traceback (most recent call last):
  File "test.py", line 16, in <module>
    column[:] = column[:]
  File ".../tables-3.0.0-py2.7-linux-x86_64.egg/tables/table.py", line
      3557, in __setitem__
    value, self.pathname)
  File ".../tables-3.0.0-py2.7-linux-x86_64.egg/tables/table.py", line
      2427, in modify_column
    mod_col[:] = column
ValueError: could not broadcast input array from shape (5,3) into shape
    (5,3,1)
```

PyTables version: 3.0.0 NumPy version: 1.7.0
I can't really see a reason why you would remove the single-dimensional dimensions. I rely on having them, and can't find a suitable workaround. Is there a particular reason for why you are doing this?

Bibliography

Journal articles

[1] Francesc Alted. 'Why modern CPUs are starving and what can be done about it'. In: *Computing in Science & Engineering* 12.2 (2010), pp. 68–71.

[4] Peter JA Cock, Tiago Antao, Jeffrey T Chang, Brad A Chapman, Cymon J Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski et al. 'Biopython: freely available Python tools for computational molecular biology and bioinformatics'. In: *Bioinformatics* 25.11 (2009), pp. 1422–1423.

[5] Alistair Cockburn and Laurie Williams. 'The costs and benefits of pair programming'. In: *Extreme programming examined* (2000), pp. 223–247.

[6] The Genomic ENCODE Consortium. 'An integrated encyclopedia of DNA elements in the human genome'. In: *Nature* 489.7414 (2012), pp. 57–74.

[8] Alan M Davis. 'Operational prototyping: A new development approach'. In: *Software, IEEE* 9.5 (1992), pp. 70–78.

[12] Jeremy Goecks, Anton Nekrutenko, James Taylor and The Galaxy Team. 'Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences'. In: *Genome Biol.* 11.8 (2010), R86.

[13] Sveinung Gundersen, Matúš Kalaš, Osman Abul, Arnoldo Frigessi, Eivind Hovig and Geir K. Sandve. 'Identifying elemental genomic track types and representing them uniformly'. In: *BMC Bioinformatics* 12.1 (2011), p. 494.

[21] Lawrence Hunter. 'Molecular biology for computer scientists'. In: *Artificial intelligence and molecular biology* (1993), pp. 23–28.

[23] Elaine R Mardis. 'A decade's perspective on DNA sequencing technology'. In: *Nature* 470.7333 (2011), pp. 198–203.

[24] Elaine R Mardis. 'Anticipating the $1,000 genome'. In: *Genome Biol* 7.7 (2006), p. 1.

[36] Geir K. Sandve, Sveinung Gundersen, Morten Johansen, Ingrid K. Glad, Krishanthi Gunathasan, Lars Holden, Marit Holden, Knut Liestøl, Ståle Nygård, Vegard Nygaard, Jonas Paulsen, Halfdan Rydbeck, Kai Trengereid, Trevor Clancy, Finn Drabløs, Egil Ferkingstad, Matúš Kalaš, Toje Lien, Morten B. Rye, Arnoldo Frigessi and Eivind Hovig. 'The Genomic HyperBrowser: an analysis web server for genome-scale data'. In: *Nucleic Acid Research* 41.W1 (2013), W133–W141.

[37] Geir K. Sandve, Sveinung Gundersen, Halfdan Rydbeck, Ingrid K. Glad, Lars Holden, Marit Holden Knut Liestøl, Trevor Clancy, Egil Ferkingstad, Morten Johansen, Vegard Nygaard, Eivind Tøstesen, Arnoldo Frigessi and Eivind Hovig. 'The Genomic HyperBrowser: inferential genomics at the sequence level'. In: *Genome biology* 11.12 (2010), R121.

[43] Stefan Van Der Walt, S Chris Colbert and Gael Varoquaux. 'The NumPy array: a structure for efficient numerical computation'. In: *Computing in Science & Engineering* 13.2 (2011), pp. 1–5.

[47] Laurie Williams, Ron Jeffries, Robert R Kessler and Ward Cunningham. 'Strengthening the case for pair programming'. In: *IEEE software* 17.4 (2000), pp. 19–25.

Other written references

[10] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal and Dana Robinson. 'An overview of the HDF5 technology suite and its applications'. In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM. 2011, pp. 36–47.

[22] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998, 248?379. ISBN: 0-201-89685-0.

[25] R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt Series. Prentice Hall/Pearson Education, 2003. ISBN: 9780135974445.

[38] Henrik Glasø Skifjeld. 'Retrieval of Genomic Data using PyTables'. MA thesis. University of Oslo, to be submitted August 2014.

[40] M. Summerfield. *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming*. Pearson Education, 2007. ISBN: 9780132703062.

Online references

[2] Francesc Alted, Ivan Vilata et al. *PyTables: Hierarchical Datasets in Python*. 2002–. URL: http://www.pytables.org/ (visited on 29/07/2014).

[3] *atexit – Exit handlers*. 2014. URL: https://docs.python.org/2/library/atexit.html (visited on 29/07/2014).

[7] *cProfile*. 2014. URL: https://docs.python.org/2/library/profile.html%5C#module-cProfile (visited on 29/07/2014).

[9] *fcntl – The fcntl and ioctl system calls*. 2014. URL: https://docs.python.org/2/library/fcntl.html (visited on 29/07/2014).

[11] *git http://git-scm.com/*. 2014. URL: http://git-scm.com/ (visited on 29/07/2014).

[14] Sveinung Gundersen, Matúš Kalaš, Osman Abul, Arnoldo Frigessi, Eivind Hovig and Geir K. Sandve. *Specification of the GTrack file format*. 2012. URL: https://hyperbrowser.uio.no/hb/static/hyperbrowser/gtrack/GTrack_specification.html?x=x (visited on 29/07/2014).

[15] Sveinung Gundersen, Geir K. Sandve and Marcin Cieslik. *GTrackCore*. 2013. URL: https://github.com/sveinugu/gtrackcore (visited on 29/07/2014).

[16] Sveinung Gundersen, Geir K. Sandve, Marcin Cieslik, Henrik Skifjeld and Brynjar Rongved. *Pytables version of GTrackCore*. 2014. URL: https://github.com/brynjagr/gtrackcore (visited on 29/07/2014).

[17] *HDF5: API Specification Reference Manual*. 2014. URL: http://www.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html (visited on 29/07/2014).

[18] *HDF5 FAQ – QUESTIONS ABOUT THE SOFTWARE*. 2014. URL: http://www.hdfgroup.org/hdf5-quest.html/%5C#gconc (visited on 29/07/2014).

[19] *HDF5 User's Guide*. 2011. URL: http://www.hdfgroup.org/HDF5/doc/PSandPDF/HDF5_UsersGuide.PDF (visited on 29/07/2014).

[20] *HDFView*. 2014. URL: http://www.hdfgroup.org/HDF5/Tutor/hdfview.html (visited on 29/07/2014).

[26] *NCSA JAVA HDF5 INTERFACE (JHI5)*. 2014. URL: http://www.hdfgroup.org/products/java/JNI/jhi5/index.html (visited on 29/07/2014).

[27] *NumExpr*. 2014. URL: https://github.com/pydata/numexpr (visited on 29/07/2014).

[28] *NumPy*. 2013. URL: http://www.numpy.org/ (visited on 29/07/2014).

[29] Jason Pellerin. *nose is nicer testing for python*. 2014. URL: http://nose.readthedocs.org/en/latest/ (visited on 29/07/2014).

[30] *PEP 8 – Style Guide for Python Code*. 2001-2013. URL: http://legacy.python.org/dev/peps/pep-0008/%5C#naming-conventions (visited on 29/07/2014).

[31] *pickle – Python object serialization*. 2014. URL: https://docs.python.org/2/library/pickle.html (visited on 29/07/2014).

[32] *Proposal for implementing column-wise tables in PyTables*. 2012. URL: https://github.com/PyTables/proposal/blob/master/column-wise-pytables.rst (visited on 29/07/2014).

[33] *PyTables User's Guide – Optimization tips*. 2013. URL: http://pytables.github.io/usersguide/optimization.html (visited on 29/07/2014).

[34] *rhdf5 – HDF5 interface to R*. 2014. URL: http://www.bioconductor.org/packages/release/bioc/html/rhdf5.html (visited on 29/07/2014).

[35] *RunSnakeRun*. 2014. URL: http://www.vrplumber.com/programming/runsnakerun/ (visited on 29/07/2014).

[39] *Structured arrays (aka "Record arrays")*. 2014. URL: http://docs.scipy.org/doc/numpy/user/basics.rec.html (visited on 29/07/2014).

[41] The HDF Group. *Hierarchical Data Format, version 5*. 1997-NNNN. URL: http://www.hdfgroup.org/HDF5/ (visited on 29/07/2014).

[42] *UCSC Data File Formats*. URL: http://genome.ucsc.edu/FAQ/FAQformat.html (visited on 29/07/2014).

[44] *WHY HDF?* 2011. URL: http://www.hdfgroup.org/why_hdf/ (visited on 29/07/2014).

[45] Wikipedia. *Compression ratio — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/wiki/Compression_ratio (visited on 29/07/2014).

[46] Wikipedia. *Distributed revision control — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/wiki/Distributed_revision_control (visited on 29/07/2014).